Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

Toni Akkala

Ohjelmistoarkkitehtuurin suunnittelu ja toteutus MEMS-antureiden demonstraatio järjestelmälle

Diplomityö
Espoo, 15. syyskuuta 2013

Valvoja: Professori Heikki Saikkonen
Ohjaaja: Filosofian maisteri Jaakko Korvenoja

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Toni Akkala		
Työn nimi:			
Ohjelmistoarkkitehtuurin suunnittelu ja toteutus MEMS-antureiden demonstraatio järjestelmälle			
Päiväys:	15. syyskuuta 2013	**Sivumäärä:**	x + 65
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	Filosofian maisteri Jaakko Korvenoja		

Murata Electronics Oy on yksi maailman johtavista piipohjaisten kapasitiivisten antureiden valmistajista. Erilaisia antureita ja anturityyppejä on lukuisia ja niitä käytetään monissa erilaisissa sovelluksissa. Asiakkaalla on yleensä tarve evaluoida uusien antureiden soveltuvuutta tiettyihin sovelluksiin. Tähän tarkoitukseen on kehitetty demonstrointi järjestelmä, joka sisältää elektroniikan, johon anturi voidaan kiinnittää sekä ohjelmiston graafisella käyttöliittymällä. Ohjelmiston tehtävänä on lukea anturilta saatava informaatio ja näyttää se asiakkaalle. Elektroniikassa olevalle mikrokontrollerille voidaan antaa komentoja nk. makrokielellä erilaisten toimintojen suorittamiseksi. Näitä toimintoja ovat esimerkiksi erilaisten anturin rekistereiden ja ulostulodatan lukeminen sekä rekistereiden kirjoittaminen kommunikointiväylän avulla.

Koska uusia antureita kehitetään jatkuvasti, tarvitaan helppo menetelmä, jolla ne voidaan lisätä demonstrointi järjestelmän ohjelmistoon. Tässä diplomityössä kehitettiin tähän tarkoitukseen soveltuva ohjelmistoarkkitehtuuri ja toteutettiin se käytännössä Java-ohjelmointikielellä. Tavoitteena oli korvata vanhat LabVIEW-ohjelmistot ja täyttää ohjelmistolle asetetut toiminnalliset ja laadulliset vaatimukset. Näiden vaatimusten perusteella sekä erilaisten suunnittelumallien avulla ohjelmistoa lähdettiin suunnittelemaan. Ohjelmistolle hahmoteltiin alustava luokkamalli, joka toimi pohjana ohjelmiston toteutukselle.

Suunnitelmaa ja toteutettua ohjelmistoa arvioitiin asetettuja toiminnallisia ja laadullisia vaatimuksia vasten. Ohjelmiston toimintaa testattiin myös erilaisilla työkaluilla sen oikean toiminnan varmistamiseksi. Sen ominaisuuksia ja resurssien käyttöä verrattiin vanhaan ohjelmistoon. Valittujen suunnittelumallien ja toteutettujen kirjastojen avulla diplomityössä onnistuttiin kehittämään hyvin laajennettava ja vakaa ohjelmisto.

Asiasanat:	MEMS-anturi, Ohjelmistoarkkitehtuuri, Java
Kieli:	Suomi

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Toni Akkala
Title:
Developing software architecture for MEMS sensors demonstration kit
Date: September 15, 2013 **Pages:** x + 65
Professorship: Software Technology **Code:** T-106
Supervisor: Professor Heikki Saikkonen
Instructor: Jaakko Korvenoja (FM)

Murata Electronics Oy is the world's leading designer and manufacturer of silicon capacitive sensors. Their products include a wide range of MEMS sensors targeted for various applications. Because a customer usually has a need to test and evaluate the operation and the performance of these sensors in different applications, a demonstration kit was designed to answer this need. The demonstration kit includes electronics with a chip carrier card to attach sensors and software with a graphical user interface. The purpose of this software is to read data from the sensor using the electronics and then show or visualize the data for the customer. Electronics is based on an embedded microcontroller which executes so-called macro kernel firmware. Specific macro language can be used to send commands to electronics to read and write sensor registers and to read output data via a digital communication bus.

Because new sensors are continuously being developed, software must have an easy method for extending the existing system and especially for defining new sensors. This Master's thesis designed and implemented software architecture for the demo kit software using Java programming language. The goal was to replace the old LabVIEW software versions and to fulfill functional and qualitative requirements. By using different design patterns and design methods, an initial class diagram was created to guide the implementation of the software.

The design and implemented software was successfully evaluated against the defined requirements. The correct operation of software was tested by using different tools. The features of the software were also reviewed against the existing software versions. This thesis achieves creating extendable and stable software by using different design patterns and implementing certain necessary software libraries.

Keywords: MEMS, Software Architecture, Java
Language: Finnish

Alkulause

Haluan kiittää valvojaani Heikki Saikkosta sekä ohjaajaani Jaakko Korvenojaa tuesta ja ohjauksesta tämän prosessin aikana.
Kaikkein suurin kiitos kuuluu vaimolleni Annalle kärsivällisyydestä ja tuesta opiskelujeni aikana.

Espoossa, 15. syyskuuta 2013

Toni Akkala

Lyhenteet

API	Ohjelmointirajapinta (Application Programming Interface)
ASIC	Sovelluskohtainen integroitu mikropiiri (Application Specific Integrated Circuit)
CPU	Prosessori (Central Processing Unit)
CSB	Signaali, jolla voidaan valita tietty elektroniikkapiiri (Chip select)
DFA	Deterministinen äärellinen automaatti (Determinstic finite Automaton)
DLL	Ajonaikaisesti linkitettävä jaettu kirjasto (Dynamic Link Library)
DOM	Ohjelmointirajapinta XML-dokumenttien sisällön käsittelemiseksi (Document Object Model)
DTD	Määrittelee XML-dokumentin tyypin (Document Type Definition)
EBNF	Notaatio konteksittomien kielioppien ilmaisuun (Extended Backus-Naur-Form)
ECS	Elektroninen jousitusjärjestelmä (Electronically Controlled Suspension)
ESC	Elektroninen ajonvakautusjärjestelmä (Electronic Stability Control)
FW	Elektronisen laitteen ohjelmistoversio (Firmware)
GPL	Lisenssi vapaiden ohjelmistojen julkaisemiseen (GNU General Public License)
GUI	Graafinen käyttöliittymä (Graphical User Interface)
HW	Elektroniikka laite (Hardware)
I^2C	Kommunikointiväylä (Inter-Integrated Circuit)
IDE	Ohjelmointiympäristö (Integrated Development Environment)

IPC	Prosessien välinen kommunikointi (Inter-Process Communication)
JAXP	Java-kielen ohjelmointirajapinta XML-dokumenttien prosessointiin (Java API for XML Processing)
JDK	Kokoelma ohjelmointi työkaluja Java-sovellusten toteuttamiseksi (Java Development Kit)
JNI	Java-kielen menetelmä natiivikoodin suorittamiseksi virtuaalikoneessa (Java Native Interface)
JNLP	Protokolla Java-sovellusten kännistämiseksi palvelimelta (Java Network Launching Protocol)
JVM	Java-kielen virtuaalikone, jossa Java-koodia suoritetaan (Java Virtual Machine)
LGPL	Lisenssi vapaiden ohjelmistojen julkaisemiseen, joka sallii tiettyjä poikkeuksia GPL lisensseihin nähden (GNU Lesser General Public License)
MEMS	Mikroelektromekaaninen järjestelmä (Mirco Electro Mechanical System)
MISO	SPI-kommunikointiväylän signaali (Master In Slave Out)
MOSI	SPI-kommunikointiväylän signaali (Master Out Slave In)
MVC	Malli-näkymä-ohjain arkkitehtuurimalli
PWM	Pulssinleveysmodulaatio (Pulse Width Modulation)
RAM	Haihtuva luku- ja kirjoitusmuisti (Random Access Memory)
RMI	Etämetodikutsu (Remote Method Invocation)
RTE	LabVIEW-ohjelmistojen suorittamiseksi tarvittava ajonaikainen kirjasto (Run-Time Engine)
SGML	Standardoitu metakieli dokumenttien merkintäkielien kuvaamiseen (Standard Generalized Markup Language)
SPI	Kommunikointiväylä (Serial Peripheral Interface)
SW	Ohjelmisto (Software)
UML	Mallinnuskieli (Unified Modeling Language)
USB	Kommunikointiväylä (Universal Serial Bus)
VCP	Virtuaalinen sarjaportti (Virtual COM port)
XML	Helposti luettava dokumenttien kuvauskieli (Extensible Markup Language)
ZIP	Eräs tiedostojenpakkausmenetelmä

Sisältö

1 Johdanto **1**

1.1 Diplomityön tavoitteet 2

1.2 Diplomityön rakenne 3

2 Lähtökohdat **4**

2.1 Sovellukset 4

2.2 Antureiden rakenne 5

2.3 Rajapintakortin kuvaus 7

2.4 Makrokieli 9

2.5 LabVIEW-käyttöliittymät 10

3 Ohjelmiston suunnittelu **13**

3.1 Vaatimusmäärittely 13

3.1.1 Toiminnalliset vaatimukset 13

3.1.2 Ei-toiminnalliset vaatimukset 14

3.2 Ohjelmistoarkkitehtuuri 15

3.3 Suunnittelumallit 17

3.3.1 Tarkkailija-suunnittelumalli 17

3.3.2 Malli-näkymä-ohjain 18

3.3.3 Asiakas-palvelin-arkkitehtuuri 19

3.4 Järjestelmän suunnittelu 20

3.4.1 Käsiteanalyysi 21

3.4.2 Käyttötapaukset 22

3.4.3 Luokkamalli 24

4 Ohjelmiston toteutus **27**

4.1 Yleiskuvaus 27
4.1.1 Hakemistorakenne 28
4.1.2 Käyttöliittymätekniikat 29
4.1.3 Sovelluslogiikka 31
4.1.4 Näkymät 32
4.1.5 Loki 33
4.2 Antureiden määrittely 34
4.2.1 Rekisterit 36
4.2.2 Ominaisuudet 37
4.3 Makrokielen jäsentäjä 38
4.4 USB-kommunikointi 41
4.4.1 JNI 42
4.4.2 JavaD2xx 43

5 Arviointi **51**

5.1 Laajennettavuus 52
5.1.1 Uuden anturityypin lisääminen 53
5.1.2 Uuden anturiperheen lisääminen 53
5.1.3 Uuden rajapintakortin lisääminen 54
5.1.4 Uuden näkymän lisääminen 54
5.2 Ohjelmiston testaus 55
5.3 Ohjelmistojen vertailua 57
5.4 Kehitys tulevaisuudessa 59

6 Yhteenveto **61**

Kuvat

1.1 Demonstraatio järjestelmä 2

2.1 SCC1300-yhdistelmäanturin lohkokaavio 5
2.2 SCC1300-anturin osittainen rekisterikartta 6
2.3 SCC1300-kiihtyvyysanturin SPI-väylän dataformaatti 6
2.4 USB-rajapintakortti 8
2.5 Anturikortteja eri anturityypeille 8
2.6 LabVIEW-ohjelmiston käyttöliittymä SCC1300-anturille . . . 11

3.1 Ohjelmiston kerrosarkkitehtuuri 16
3.2 Tarkkailija-suunnittelumalli 17
3.3 Malli-näkymä-ohjain 18
3.4 Suunniteltavan järjestelmän yleiskuva 20
3.5 Käyttötapauskaavio 22
3.6 Luokkamalli 24

4.1 Ohjelmiston hakemistorakenne 30
4.2 Sovelluslogiikan kuvaus 32
4.3 AccChartViewController 33

5.1 JConsole-työkalun yleisnäkymä 56
5.2 JConsole-työkalun "*CMS Old Gen*"-näkymä 57
5.3 MFIDemo-ohjelmiston näkymä ulostulodatalle 58

Listaukset

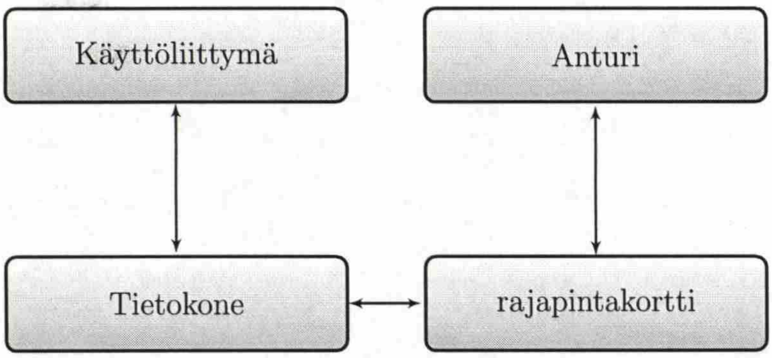
4.1 LOGGER-olion luominen 34
4.2 LOGGER-olion alustus 34
4.3 Osa demos.xml tiedostoa 35
4.4 Rekistereiden määrittely 37
4.5 SCC1300-D04.properties 38
4.6 Makrokielen merkistö 39
4.7 Makrokielen päätesymbolit 39
4.8 Makrokielen produktioita 40
4.9 JNI-esimerkki 43
4.10 *FT_DEVICE_LIST_INFO_NODE*-tietue 44
4.11 *DeviceInfoListNode*-luokka 44
4.12 *JavaD2xxException*-luokka 44
4.13 *JavaD2xx*-luokan metodeja. 45
4.14 *open()*-funktion prototyyppi 45
4.15 *open()*-funktion toteutus 46
4.16 *read()*-funktion toteutus 47
4.17 ID:iden puskurointi 48
4.18 Natiivikirjaston lataus 49
5.1 JNLP-esimerkki 60

Luku 1

Johdanto

Murata Electronics Oy kehittää ja valmistaa MEMS-teknologiaan (Mirco Electro Mechanical System) perustuvia kiihtyvyys-, kallistus- ja kulmanopeusantureita hyvin moniin erilaisiin sovelluksiin. Esimerkiksi autoteollisuudessa, antureita käytetään elektronisiin ajonvakausjärjestelmiin (Electronic Stability Control, ESC) sekä elektronisiin jousitusjärjestelmiin (Electronically Controlled Suspension, ECS). Autoteollisuuden lisäksi sovelluksia löytyy myös kulutuselektroniikasta, terveysteknologioista, instrumenteista, työkoneista ja ilmailuteollisuudesta [14]. Anturin ominaisuudet määrittelevät sen soveltuvuuden tiettyihin sovelluksiin. Uusien anturiperheiden (sensor family) tai anturityyppien valmistuessa on tärkeää, että niitä päästään demonstroimaan asiakkaille mahdollisimman aikaisessa vaiheessa. Tällä pyritään takaamaan asiakkaan kiinnostus uuteen anturiin sekä nopeuttamaan anturin pääsyä markkinoille. Asiakas haluaa yleensä testata ja evaluoida anturia mittaamalla anturin vastetta tai anturin kohinaa tai kokeilemalla sen muita ominaisuuksia. Tähän tarkoitukseen on kehitetty kuvassa 1.1 näkyvä demonstraatio järjestelmä (demo kit), joka sisältää rajapintakortin (USB Interface card) sekä ohjelmiston graafisella käyttöliittymällä. Rajapintakorttiin voidaan kiinnittää erilaisia antureita ja se tarjoaa digitaalisen kommunikointiväylän antureille. Ohjelmistolla voidaan siten kommunikoida anturin kanssa, esitellä anturin ominaisuuksia ja toimintaa asiakkaille visualisoimalla anturilta luettavaa dataa eri tavoin.

LabVIEW-ohjelmointiympäristöllä on vankka jalansija antureiden tuotantotestauksessa ja aikaisemmin myös demonstraatio järjestelmän ohjelmistot on toteutettu kyseisellä ohjelmointiympäristöllä. Tästä on kuitenkin aiheutunut erilaisia haasteita. Esimerkiksi, jotta LabVIEW:llä käännettyjä ohjelmia voidaan suorittaa itsenäisinä sovelluksina, tarvitaan RTE-kirjasto (Run-Time



Kuva 1.1: Demonstraatio järjestelmä

Engine), joka sisältää suorituksen aikana tarvittavat kirjastot sekä tiedostot. LabVIEW:n RTE joudutaan yleensä asentamaan erikseen, koska harvoilla käyttäjillä se on valmiiksi asennettuna. RTE:n sisällyttäminen demonstraatio järjestelmän asennustiedostoihin kasvattaa myös asennuspaketin kokoa merkittävästi. Eri antureille tehtyjen ohjelmistojen rakenne poikkeaa myös toisistaan, mikä on vaikeuttanut näiden ohjelmistojen ylläpitoa. Näistä syistä johtuen erilaisille antureille päätettiin suunnitella ja toteuttaa yksi yhteinen ohjelmisto Java-ohjelmointikielellä. Java-kielen tulkki (Java Runtime Engine, JRE) on yleensä kaikilla käyttäjillä valmiiksi asennettuna. Java-kieli tarjoaa myös suuren määrän avoimen lähdekoodin kirjastoja, esimerkiksi kuvaajien piirtämiseen tai 3D-grafiikkaan, hyvän tuen ohjelmien käyttöliittymien koristeluun ja kansainvälistämiseen sekä merkkijonojen käsittelyyn esimerkiksi säännöllisten lausekkeiden avulla. Java-ohjelmointikieltä voidaan käyttää myös esimerkiksi Android-sovelluksissa, joten ohjelmistoa varten suunniteltuja komponentteja voidaan hyödyntää myös tällaisten sovellusten kanssa.

1.1 Diplomityön tavoitteet

Tämän diplomityön tavoitteena on kuvata ohjelmistoarkkitehtuuri sekä suunnitella ja toteuttaa demonstraatio järjestelmää varten ohjelmisto, jolla voidaan korvata olemassa olevat LabVIEW-ohjelmistot. Työssä asetetaan vaatimukset ohjelmistolle, kuvataan ohjelmistoon liittyvät komponentit sekä niiden toteutus ja pohditaan, kuinka hyvin valituilla ratkaisuilla voidaan täyttää asetetut vaatimukset.

1.2 Diplomityön rakenne

Luvussa 2 kuvataan lähtökohdat, esimerkkisovellukset antureille, demonstrointi järjestelmässä käytettävä rajapintakortti sekä makrokieli, jolla rajapintakortille annetaan komentoja ja luetaan dataa anturilta. Luvussa 3 määritellään vaatimukset toteutettavalle ohjelmistolle ja käydään läpi suunniteltavaan arkkitehtuuriin liittyviä yksityiskohtia. Luvussa 4 kuvataan ohjelmiston toteutus ja käytettävät kirjastot. Luvussa 5 arvioidaan suunnitelmaa sekä toteutusta ja luvussa 6 esitetään johtopäätökset.

Luku 2

Lähtökohdat

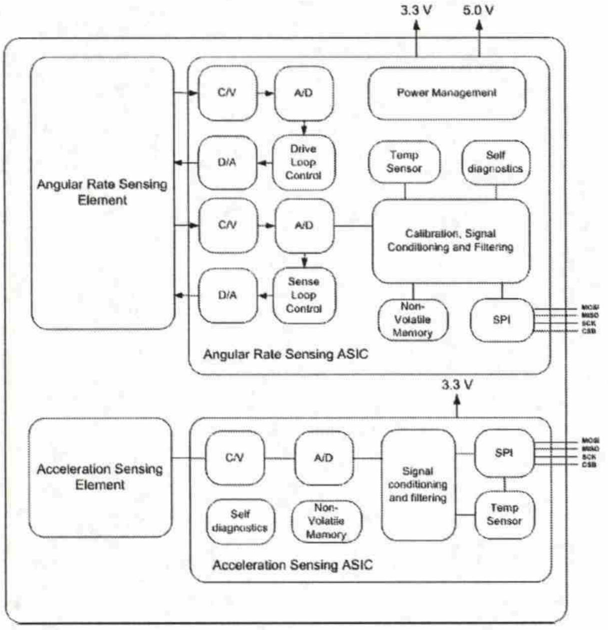
Tämä luku kertoo ensin erilaisista antureita käyttävistä sovelluksista ja yleisesti antureiden rakenteesta. Sen jälkeen kuvataan demonstraatio järjestelmän rajapintakortin toteutus sekä makrokieli, jolla annetaan komentoja rajapintakortille esimerkiksi rekistereiden lukemiseksi anturilta. Lopuksi kuvataan hieman olemassa olevia ohjelmistoja, jotka on toteutettu LabVIEW-ohjelmointiympäristöllä.

2.1 Sovellukset

Antureiden pieni koko ja hinta ovat mahdollistaneet niiden käyttämisen hyvin monissa erilaisissa sovelluksissa. Autoteollisuudessa yksi tärkeimmistä sovelluksista on elektroninen ajonvakautusjärjestelmä. Siinä järjestelmä pyrkii korjaamaan auton ali- tai yliohjautuvuutta esimerkiksi jarruttamalla yksittäisillä pyörillä antureilta saatavan informaation mukaan [2]. Toinen yleistynyt sovellus on elektroninen jousitusjärjestelmä, jossa auto mukauttaa jousitusta ajo-olosuhteiden mukaan. Muita mielenkiintoisia sovelluksia ovat esimerkiksi sydämentahdistimet sekä erilaiset ihmisen sykettä mittaavat sovellukset, kuten henkilövaaka, jossa kiihtyvyysanturia käytetään ballistogardiografisiin mittauksiin [18]. Tampereen teknillinen yliopisto hyödynsi tutkimuksessaan kulmanopeusanturia maapallon pyörimisen mittaamiseksi [12]. Erilaiset antureiden ominaisuudet, kuten mittausalue, herkkyys tai kohina, määrittelevät millaisiin erilaisiin sovelluksiin kyseinen anturi soveltuu.

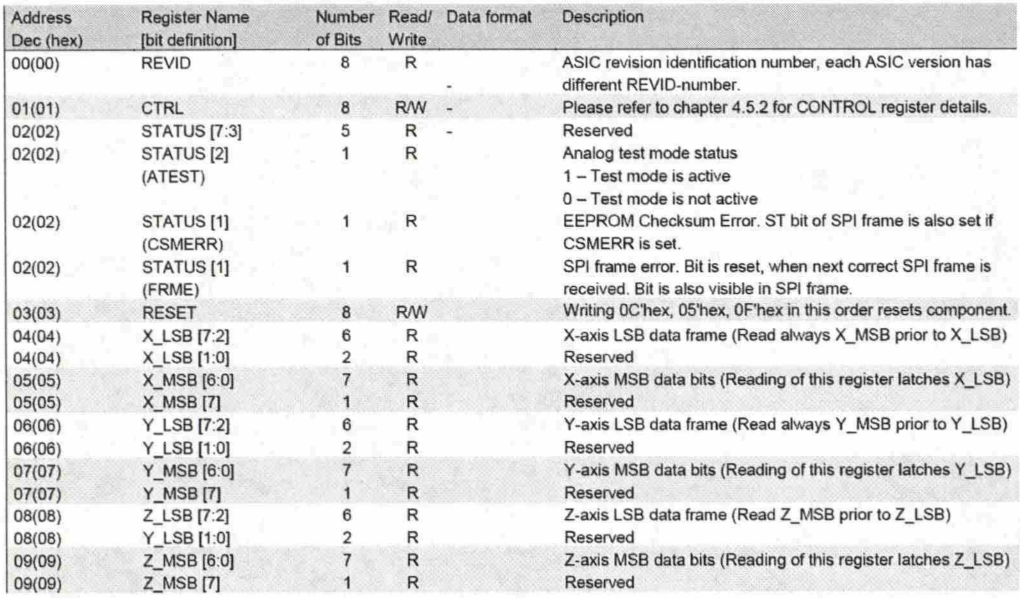
2.2 Antureiden rakenne

Anturi on komponentti, joka koostuu anturielementistä, integroidusta mittauselektroniikasta eli ASIC-piiristä (Application Specific Integrated Circuit) sekä ympäristövaikutuksilta suojatusta kotelosta. Kiihtyvyys aiheuttaa anturielementin sisäisten massojen liikkumisen, joka saa aikaan kapasitanssin muutoksen anturielementissä. ASIC-piirin tehtävänä on mitata tätä muutosta. Analogisilla antureilla ASIC-piiri muuttaa tämän kapasitanssin muutoksen jännitteeksi, joka voidaan mitata anturin nastoista (pin). Digitaalisilla antureilla kapasitanssin muutos muutetaan digitaaliseksi numeroarvoksi. ASIC-piiriltä voidaan lukea nämä ulostuloarvot digitaalisen SPI- (Serial Peripheral Interface) tai I^{2}C-väylän (Inter-Integrated Circuit) avulla. Tässä luvussa käytetään esimerkkinä SCC1300-yhdistelmäanturia, jonka lohkokaavio on esitetty kuvassa 2.1. SCC1300-anturi on yleisesti käytetty esimerkiksi ESC-järjestelmissä.



Kuva 2.1: SCC1300-yhdistelmäanturin lohkokaavio

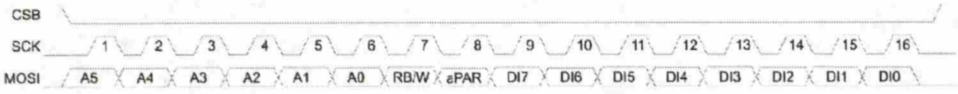
SCC1300 on yhdistelmäanturi, jossa on kiihtyvyysanturi ja kulmanopeusanturi samassa kotelossa. Molemmilla antureilla on oma mittauselementtinsä sekä oma ASIC-piirinsä. Siksi siinä on myös kaksi erillistä SPI-väylää, kummallekin ASIC-piirille omansa [13]. Kaikissa yhdistelmäantureissa näin ei ole, vaan kotelossa voi olla vaan yksi ASIC-piiri, jolta luetaan sekä kiihtyvyysanturin että kulmanopeusanturin ulostulot. Kuvassa 2.2 on yhdistelmäanturin kiihtyvyysanturin ASIC-piirin osittainen rekisterikartta, josta voidaan nähdä rekisteriosoitteet ja kuvaukset esimerkiksi CTRL-rekisterille ja kiihtyvyysanturin x-, y- ja z-akseleiden ulostuloille.



Address Dec (hex)	Register Name [bit definition]	Number of Bits	Read/ Write	Data format	Description
00(00)	REVID	8	R	-	ASIC revision identification number, each ASIC version has different REVID-number.
01(01)	CTRL	8	R/W	-	Please refer to chapter 4.5.2 for CONTROL register details.
02(02)	STATUS [7:3]	5	R	-	Reserved
02(02)	STATUS [2] (ATEST)	1	R		Analog test mode status 1 – Test mode is active 0 – Test mode is not active
02(02)	STATUS [1] (CSMERR)	1	R		EEPROM Checksum Error. ST bit of SPI frame is also set if CSMERR is set.
02(02)	STATUS [1] (FRME)	1	R		SPI frame error. Bit is reset, when next correct SPI frame is received. Bit is also visible in SPI frame.
03(03)	RESET	8	R/W		Writing 0C'hex, 05'hex, 0F'hex in this order resets component.
04(04)	X_LSB [7:2]	6	R		X-axis LSB data frame (Read always X_MSB prior to X_LSB)
04(04)	X_LSB [1:0]	2	R		Reserved
05(05)	X_MSB [6:0]	7	R		X-axis MSB data bits (Reading of this register latches X_LSB)
05(05)	X_MSB [7]	1	R		Reserved
06(06)	Y_LSB [7:2]	6	R		Y-axis LSB data frame (Read always Y_MSB prior to Y_LSB)
06(06)	Y_LSB [1:0]	2	R		Reserved
07(07)	Y_MSB [6:0]	7	R		Y-axis MSB data bits (Reading of this register latches Y_LSB)
07(07)	Y_MSB [7]	1	R		Reserved
08(08)	Z_LSB [7:2]	6	R		Z-axis LSB data frame (Read Z_MSB prior to Z_LSB)
08(08)	Y_LSB [1:0]	2	R		Reserved
09(09)	Z_MSB [6:0]	7	R		Z-axis MSB data bits (Reading of this register latches Z_LSB)
09(09)	Z_MSB [7]	1	R		Reserved

Kuva 2.2: SCC1300-anturin osittainen rekisterikartta

SCC1300-anturin kahden ASIC-piirin SPI-väylien dataformaatit poikkeavat toisistaan. Kiihtyvyysanturin SPI-väylä käyttää 8-bittisiä osoitteita, kun taas kulmanopeusanturin SPI-väylä käyttää 16-bittisiä osoitteita.



Kuva 2.3: SCC1300-kiihtyvyysanturin SPI-väylän dataformaatti

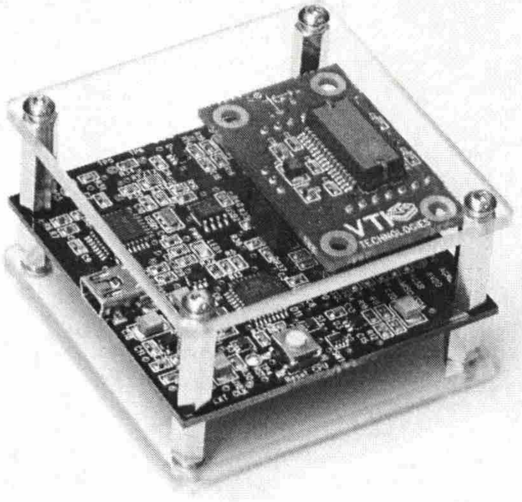
Kuvassa 2.3 on kiihtyvyysanturin SPI-väylän dataformaatti. Jokainen kommunikointikehys on 16-bittiä pitkä. SPI-väylän isäntänä toimii mikrokontrolleri ja orjana anturi. Isäntä valitsee anturin CSB-signaalilla (Chip Select), minkä jälkeen dataa kellotetaan jokaisella kellopulssilla. Ensimmäiset 8 MOSI-bittiä (Master Out Slave In) sisältävät informaation osoitteesta, operaatiosta sekä pariteetista. Lukuoperaatiossa seuraavilla 8 MOSI-bitillä ei ole merkitystä, mutta kirjoitusoperaatiossa seuraavat 8 bittiä sisältävät kirjoitettavan datan. Ensimmäiset 8 MISO-bittiä (Master In Slave Out) sisältävät tila-bittejä (status bit), muutaman ylimääräisen bitin sekä pariteetin. Jälkimmäiset 8 bittiä sisältävät luettavan datan.

Antureiden ominaisuudet vaihtelevat eri antureiden välillä. Alla on esitetty muutamia suunniteltavalle ohjelmistolle merkittäviä anturin ominaisuuksia. Tiukat vaatimukset esimerkiksi autoteollisuudessa ovat edesauttaneet erilaisten vikadiagnostiikka ominaisuuksien kehittämisessä anturin oikean toiminnan varmistamiseksi. Sovelluksissa, joissa anturi on kiinnitetty mukana kulkevaan laitteeseen, täytyy yleensä pyrkiä minimoimaan anturin virrankulutusta. Useissa antureissa on erilaisia toimintatiloja (operation mode), joilla voidaan pienentää virrankulutusta esimerkiksi laittamalla anturi hetkellisesti lepotilaan (standby mode).

- Akseleiden lukumäärä
- Akseleiden mittaussuunnat (x, y, z)
- Mittausalue (g tai °)
- Herkkyys (LSB/g, V/g tai LSB °/s)
- ODR (Output Data Rate)
- Kommunikointiväylän taajuus
- Virranhallinta ominaisuudet
- Vikadiagnostiikka ominaisuudet
- Käyttöjännite

2.3 Rajapintakortin kuvaus

Demonstraatio järjestelmän rajapintakortti näkyy kuvassa 2.4. Rajapintakortti pohjautuu Atmel:n ATMega168V-mikrokontrolleriin, joka hoitaa kommunikoinnin anturin ja ohjelmiston välillä. Mikrokontrollerin SPI-väylistä on kytkennät anturille, joka kiinnitetään erillisellä anturikortilla (chip carrier) rajapintakorttiin. Rajapintakortti saa käyttöjännitteensä suoraan USB-väylästä (Universal Serial Bus), joten ulkoista virtalähdettä ei tarvita, jos anturin käyttöjännite on alle 5 volttia. Ohjelmistolle näkyvä liitäntä rajapintakortissa on Future Technology Devices International Limited:n (FTDI) valmistama FT232BM USB UART-piiri, joka mahdollistaa sarjamuotoisen kommunikoinnin tietokoneen ja mikrokontrollerin välillä USB-väylän kautta. Asentamalla FTDI:n laiteajurit piiri näkyy ohjelmistolle suoraan virtuaalisena sarjaporttina (Virtual COM port, VCP) tai USB-laitteena. Tämän työn



Kuva 2.4: USB-rajapintakortti

suunnittelun aikana oli suunnitteilla seuraavan sukupolven rajapintakortti, jossa on Atmel:n SAM3S ARM Cortex-M3 mikrokontrolleri ja FTDI:n FT245 USB-FIFO-piiri.

(a) SCC1300 (b) CMA3000 (c) SCA8X0/21X0/3100

Kuva 2.5: Anturikortteja eri anturityypeille

Mikrokontrollerille on toteutettu C-kielinen ohjelma, jonka toiminnallisuutta on kuvattu luvussa 2.4. Lähettämällä makrokomentoja rajapintakortille esimerkiksi terminaalista, sitä voidaan käyttää myös ilman demonstraatio järjestelmän ohjelmistoa. Kuvassa 2.5 näkyy erilaisia rajapintakorttiin liitettäviä anturikortteja, joissa anturikomponentti on juotettu pirilevylle.

2.4 Makrokieli

Makrokieli on anturin nastojen ja kommunikointiväylän (SPI / I^2C) konfigurointiin ja ohjaukseen suunniteltu komentokieli, jota käytetään demonstraatio järjestelmään suunnitellussa rajapintakortissa [10]. Makrokielestä on olemassa useita versioita, jotka eroavat hieman toisistaan niin tuettujen makrojen tai komentojen kuin komentoihin saatavien vastaustenkin osalta. Mikrokontrollerin sulautettu ohjelmisto (Firmware, FW) suorittaa ns. makrokerneliä, jolle voidaan antaa joko konfigurointi- tai datasiirtokomentoja. Mikrokontrolleri tallentaa luodut makrot RAM-muistiin (Random Access Memory). Muistiin mahtuu useampia makroja yhtä aikaa, mutta vain yksi makro voi olla suoritusvuorossa ja suoritettavan makron voi vaihtaa. Makroa suoritettaessa voidaan määritellä suorituskertojen lukumäärä. Tulokset tallennetaan makrokernelin rekistereihin, mistä ne voidaan lukea erikseen toisilla komennoilla.

Taulukko 2.1: Makrokielen konfigurointikomennot

Komento	**Kuvaus**
*10	Suorittaa suoritusvuorossa olevan makron.
*20	Listaa muistissa olevat makrot.
*21	Luo uuden makron annetulla nimellä.
*22	Vaihtaa suoritusvuorossa olevan makron ja mahdollisesti suorittaa sen.
*23	Tulostaa makron sisällön.
*24	Luo makron, suorittaa sen ja lopuksi poistaa makron muistista.
*29	Poistaa halutun makron tai kaikki makrot (29A).
*80	Tulostaa rajapintakortin tiedot (SW, HW, FW).
*81	Kuten *80, mutta tulostaa lyhyen version tiedoista.

Makrokielen syntaksi on melko yksinkertainen. Konfigurointikomento alkaa '*'-merkillä, jonka perään liitetään komennon kaksinumeroinen indeksi. Jos komentoon liittyy suoritettavia datasiirtokomentoja, ne voidaan listata komentoon puolipilkulla erotettuina. Datansiirtokomennoilla on vaihteleva määrä parametreja komennosta riippuen. Konfigurointikomento päättyy "#"-merkkiin. Jos esimerkiksi halutaan luoda makrokomento nimeltään *setPin*, joka muuttaa IO-nastan numero 4 tilan 1:ksi, annetaan komento: **21,setPin,POUT,4,1;#*. Kun komento on onnistuneesti lähetetty rajapintakortille, voidaan makro suorittaa komennolla **10,c1#*, joka suorittaa makron yhden kerran (c1). Taulukoissa 2.1 ja 2.2 on listattu konfigurointi- ja datansiirtokomennot ja niiden kuvaukset.

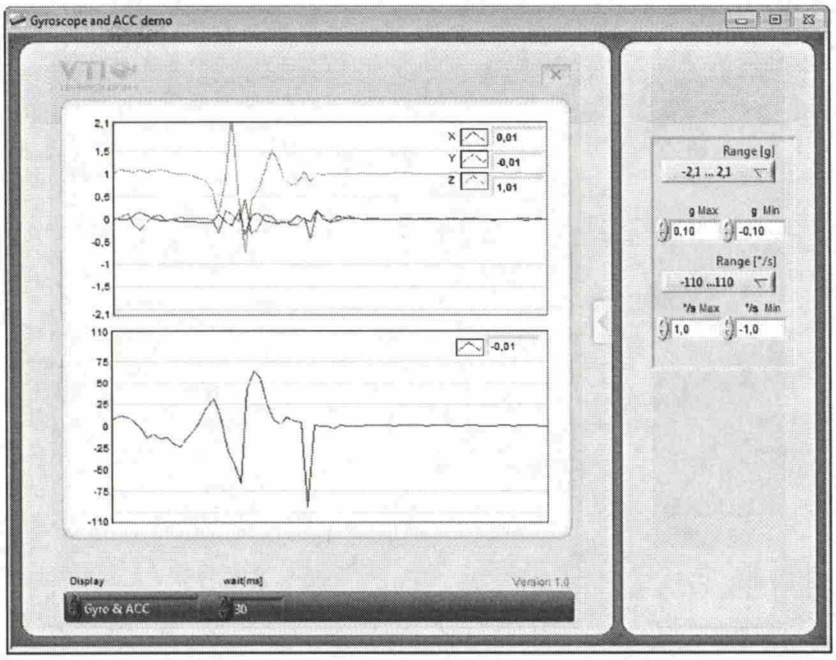
Taulukko 2.2: Makrokielen datansiirtokomennot

Komento	**Kuvaus**
ADD	Laskee kahden rekisterin sisällön summan.
I2C	Määrittelee I^2C osoitteen, kirjoittaa tai lukee rekisterin käyttäen I^2C-väylää.
NONE	Ei tee mitään.
PIN	Lukee anturin nastan tilan.
PINWAIT	Odottaa, kunnes anturin nastan tila vastaa määriteltyä tilaa.
POUT	Asettaa nastan tilan (0 = GND / 1 = VCC).
PWR	Asettaa anturin käyttöjännitteen.
REPEAT	Voidaan käyttää silmukan luomiseen, jossa toistetaan muita komentoja.
SEND	Lähettää edellisen SPI-väylän datan mikrokontrollerilta ohjelmistolle.
SENDSTR	Lähettää annetun merkkijonon.
SPI	Lukee tai kirjoittaa anturin rekisterin SPI-väylää käyttäen.
TIMESTAMP	Palauttaa laskurin arvon, jota voidaan käyttää aikaleimana.

2.5 LabVIEW-käyttöliittymät

LabVIEW-käyttöliittymiä on toteutettu melkein kaikille erilaisille antureille paineantureista kulmanopeusantureihin. Kuvassa 2.6 on esimerkki SCC1300-yhdistelmäanturin demonstraatio järjestelmän ohjelmistosta. Ohjelmistossa on mahdollista tarkastella kuvaajia kiihtyvyysanturille ja kulmanopeusanturille sekä niiden numeerisia arvoja. Käyttäjä voi säätää kuvaajien raja-arvoja, joko valitsemalla suoraan halutut raja-arvot tai syöttämällä tietyt minimi ja maksimi arvot niille varattuihin kontrolleihin.

LabVIEW-käyttöliittymissä on eroja toiminnallisuuksissa eri ohjelmaversioiden välillä. Tässä kuvattu esimerkki sisältää yhden yksinkertaisimmista toiminnallisuuksista. Muita LabVIEW-käyttöliittymiin toteutettuja toiminnallisuuksia tai ominaisuuksia ovat:



Kuva 2.6: LabVIEW-ohjelmiston käyttöliittymä SCC1300-anturille

- Bubble view - Näyttää anturin kallistuksen asteina.
- 3D-malli - Liikuttaa 3D-mallia anturin liikkeiden mukaan.
- Rekistereiden konfigurointi - Käyttäjä voi valita rekisterit ja lukea rekistereitä tai kirjoittaa dataa rekistereihin.
- SPI-kehyksien mallinnus - Näyttää edellisen operaation SPI-kehyksen aaltomuodot.
- PWM-simulaatio - Simuloi anturin generoimaa PWM-signaalia (Pulse Width Modulation).

Uutta ohjelmistoa suunniteltaessa täytyi käydä läpi vanhojen ohjelmistojen ominaisuuksia ja päättää, mitkä niistä toteutetaan myös uuteen ohjelmistoon. Osa pois jätettävistä ominaisuuksista oli tarkoitettu pelkästään vanhoille tuotannosta poistuneille analogisille tuotteille. Tällaisia ominaisuuksia ei ole mielekästä toteuttaa uuteen ohjelmistoon. Ohjelmien toiminnassa on myös ollut tiettyjä ongelmia. Käyttöliittymät ovat aiemmin kommunikoineet rajapintakortin kanssa virtuaalisarjaportin kautta, joka käyttäjän on valittava ohjelman käynnistyessä. Käyttäjille ei välttämättä ole ollut selvää, mitä porttia käyttää, mikäli tietokoneessa on ollut samaan aikaan kytkettynä useita laitteita. Myös toistuvasti lähetettävät pienet määrät dataa ovat aiheuttaneet ongelmia Windows-käyttöjärjestelmän virtuaalisarjaportin ajureiden kanssa. Jotta ohjelmat toimisivat oikein, käyttäjän on täytynyt pienentää virtuaalisarjaportin latenssi aikaa (latency timer) [11] sarjaportin asetuksista. Näistä käyttäjälle välttämättömistä ylimääräisistä operaatioista halutaan päästä eroon uudessa ohjelmistoarkkitehtuurissa. Tästä syystä rajapintakorttia tullaan käyttämään suoraan USB-laitteena, jolloin latenssi aikaa ja muita parametreja voidaan tarvittaessa muuttaa ohjelmallisesti käyttäen FTDI:n ohjelmointirajapintaa [5]. Rajapinnan avulla voidaan myös näyttää enemmän tietoja laitteesta, johon yhteys halutaan muodostaa. Tällaisia tietoja voivat olla esimerkiksi laitteen kuvaus tai sarjanumero.

Luku 3

Ohjelmiston suunnittelu

Tässä diplomityössä on siis tarkoituksena suunnitella ohjelmistoarkkitehtuuri demonstraatio järjestelmän ohjelmistolle sekä toteuttaa se Java-ohjelmointikielellä. Ohjelmiston tulee toimia eri käyttöjärjestelmissä ja sen tulee korvata kaikki aiemmat LabVIEW-ohjelmistot. Ohjelmistolla tarkoitetaan tässä yhteydessä itse ohjelmaa sekä kaikkia sen tarvitsemia hakemistoja ja tiedostoja eli kokonaisuutta, joka toimitetaan asiakkaalle. Tässä luvussa kuvataan vaatimusmäärittely, jonka jälkeen suunnitellaan ohjelmiston arkkitehtuuria ja ohjelmiston komponenttien ositusta muodostamalla luokkamalli käyttäen apuna erilaisia suunnittelumalleja.

3.1 Vaatimusmäärittely

3.1.1 Toiminnalliset vaatimukset

Ennen ohjelmiston suunnittelua sille asetettiin tiettyjä vaatimuksia, jotka sen tulisi täyttää. Nämä vaatimukset tulevat osin asiakkailta ja osin vanhojen toteutusten pohjalta. Toiminnalliset vaatimukset kuvaavat sitä miten ohjelman tulisi toimia. Toiminnalliset vaatimukset toteutettavalle ohjelmistolle ovat seuraavat:

- Järjestelmän käynnistyessä käyttäjä voi valita käytettävän rajapintakortin sekä muodostaa siihen yhteyden ja valita listalta käytettävän anturin ja anturin tyypin.
- Yhteyden muodostamisen jälkeen järjestelmä voi lähettää ja vastaanottaa dataa anturilta.

- Järjestelmä osaa muodostaa makrokomentoja.
- Järjestelmä konfiguroi anturin ja aloittaa ulostulodatan vastaanottamisen automaattisesti ilman käyttäjältä vaadittavia toimenpiteitä.
- Ohjelmiston käyttöliittymä sisältää erilaisia näkymiä, joissa vastaanotettua dataa voidaan visualisoida eri tavoin, dataa voidaan esimerkiksi näyttää kuvaajassa tai hyödyntää muuten, esimerkiksi vuorovaikutteisen 3D-ympäristön kanssa.
- Käyttäjällä on mahdollisuus tallentaa dataa tiedostoon raaka datana eli suoraan rajapintakortilta vastaanotettu data tai muunnettuna lukuarvoksi eli kiihtyvyydeksi tai muuksi anturityypille sopivaksi arvoksi.
- Ohjelmiston käyttöliittymä sisältää näkymän, jossa käyttäjä voi kirjoittaa ja lukea määriteltyjä anturin rekistereitä ja nähdä näiden rekistereiden tarkemmat kuvaukset.
- Ohjelmiston käyttöliittymä sisältää näkymän, jossa käyttäjä voi itse kirjoittaa makrokomentoja ja lähettää niitä rajapintakortille.
- Järjestelmä osaa tarkistaa käyttäjän antamien makrokomentojen syntaksin ja semantiikan jäsentäjän avulla.
- Järjestelmä tarjoaa etärajapinnan anturilta saatavaan dataan, jolloin yhdeltä anturilta vastaanotettua dataa voidaan hyödyntää eri virtuaalikoneessa ajettavan prosessin kanssa. Toiminnallisuus otetaan huomioon suunnittelussa, mutta sen toteutus rajataan tämän työn ulkopuolelle.
- Käyttäjällä on mahdollisuus tarkastella anturin datalehtiä käytettävän anturin mukaan.

3.1.2 Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset kuvaavat enemmän ohjelman ominaisuuksia ja sitä miten toiminnalliset vaatimukset tulee täyttää. Kehitettävän ohjelmiston ei-toiminnallisia vaatimuksia ovat:

- Ohjelmiston toteutus tehdään Java-ohjelmointikielellä, ohjelmiston tulee toimia Java SE 6 tai uudemmalla versiolla, ohjelmoinnissa tulee noudattaa tiettyjä ohjelmointikäytäntöjä (code conventions).
- Ohjelmiston käyttöliittymän toteutus Swing-komponenttikirjastolla. Ohjelmiston suunnittelussa on huomioitava, että käyttöliittymän toteutus tulee olla vaihdettavissa, toteutus esimerkiksi Java FX -alustalla.
- Ohjelmiston tulee toimia kaikilla olemassa olevilla rajapintakorteilla.

- Rajapintakortin tulisi toimia USB-väylän kautta ilman virtuaalista sarjaporttia eli käyttäen FTDI:n D2XX-rajapintaa.
- Ohjelmiston tulisi toimia luotettavasti, ohjelman suoritus ei saa kaatua eikä kommunikointi rajapintakortin kanssa saa kadottaa tietoa.
- Ohjelmisto tulee olla helposti ylläpidettävä ja laajennettava. Uusien anturiperheiden ja anturityyppien lisääminen ohjelmistoon tulisi olla mahdollisimman yksinkertaista ja helppoa.
- Ohjelmisto tulee olla siirrettävissä ainakin viimeisemmille Windows- ja Linux-käyttöjärjestelmille.
- Ohjelmiston asennus tulee olla mahdollisimman yksinkertaista eri käyttöjärjestelmille.
- Järjestelmä kirjoittaa virhetilanteiden tiedot lokitiedostoon.
- Käyttöliittymän kieli on mahdollista vaihtaa.

3.2 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuuri kuvaa ohjelmiston keskeiset osat ja niiden keskinäiset suhteet sekä osien suhteet muuhun ympäristöön. Suhteet voidaan kuvata niin staattisina kuin dynaamisinakin. Jos ohjelmistoarkkitehtuuri on monimutkainen, arkkitehtuuria voidaan tarkastella pienemmissä osissa eri näkökulmista. Tällä tavalla laajan järjestelmän eri järjestelmien osien hahmottaminen voi olla helpompaa. [8]

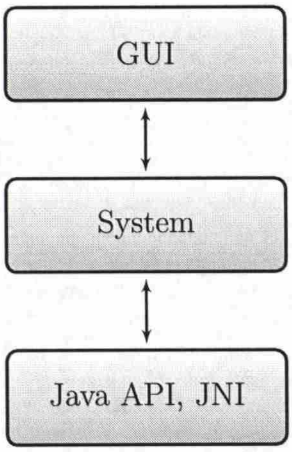
Hyvä arkkitehtuuri mahdollistaa ohjelmiston inkrementiaalisen ja rinnakkaisen kehityksen sekä helpottaa ohjelmiston testausta ja ylläpitoa. Arkkitehtuuritason ratkaisuilla pyritään siis ohjelmiston hyvään ylläpidettävyyteen ja laajennettavuuteen sekä suunniteltavien komponenttien uudelleenkäytettävyyteen muissa yhteyksissä. Arkkitehtuuri perustuu ohjelmiston dekompositioon, eli ohjelmiston pilkkomiseen komponentteihin valittujen perusteiden mukaan. Arkkitehtuuri kuvaa siis komponentit sekä niiden suhteet muihin komponentteihin eli niiden välisen rajapinnan. [8]

Rajapinta määrittelee sen, miten komponentit kommunikoivat toistensa kanssa. Rajapinnan tehtävänä on erottaa toisistaan komponentin tarjoama palvelu ja sen toteutus. Komponentti, joka tarjoaa palvelua, tarjoaa eli toteuttaa kyseisen rajapinnan. Palvelua käyttävä komponentti vaatii kyseisen rajapinnan. [8]

Ohjelmiston dekompositio tehdään tässä työssä käsitemallin avulla. Ensin mietitään ongelman kohdealueen käsitteistöä vaatimusten avulla, josta saadaan valittua ohjelmiston kannalta olennaiset käsitteet. Näiden käsitteiden ja käyttötapauskuvauksien avulla voidaan hahmotella järjestelmään toteutettavia komponentteja. Luvussa 3.3 esitellään muutamia suunniteltavalle arkkitehtuurille olennaisia suunnittelumalleja.

Työssä suunniteltavan ohjelmiston arkkitehtuuri perustuu kerrosarkkitehtuuriin (layered architecture), jossa graafinen käyttöliittymä (Graphical User Interface, GUI) on erotettu sovelluslogiikasta (kts. 3.3.2). Kuvassa 3.1 nähdään, miten ohjelmisto kuvataan kerroksina, joissa ylemmät kerrokset käyttävät alemman kerroksen tarjoamia palveluita rajapintojen kautta. Käyttöliittymä on yksi kerros, joka käyttää alemman kerroksen palveluita rajapinnan läpi, tässä tapauksessa sovelluslogiikan tarjoamia palveluita. Myös sovelluslogiikan alla on kerros, Java API (Application Programming Interface), joka tarjoaa palveluita sovelluslogiikalle, esimerkiksi tiedostojärjestelmän operaatiot tiedostojen kirjoittamista varten. Arkkitehtuuri voi kuvata kerrokset lähtien ohjelmiston käyttäjästä aina fyysiseen anturiin asti. Arkkitehtuurin käyttäminen mahdollistaa esimerkiksi käyttöliittymäkerroksen vaihtamisen toiseen toteutukseen.

Sovelluslogiikan (System) tehtävänä on tarjota palvelut järjestelmän ja anturin kontrolloimiseksi. Käyttöliittymän tehtävänä on tarjota palvelut anturilta saatavan datan visualisoimiseksi sekä sovelluslogiikan ohjaamiseksi määritellyn rajapinnan avulla. Sovelluslogiikalla on mahdollisuus tehdä takaisinkutsuja käyttöliittymään toisen rajapinnan kautta, esimerkiksi ilmoittaakseen erilaisista virhetilanteista.



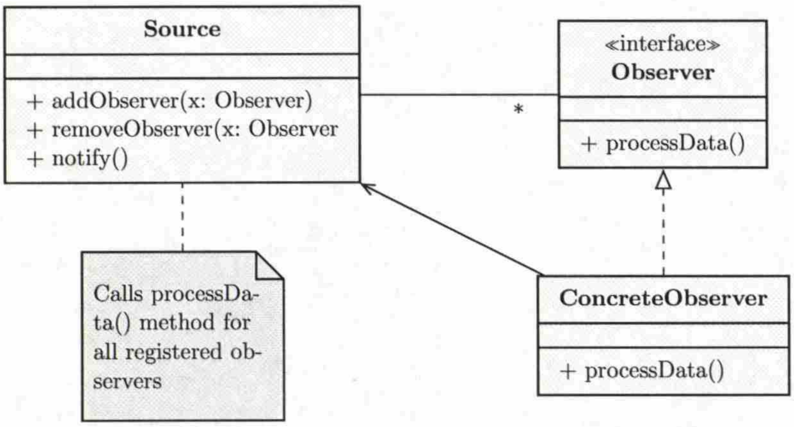
Kuva 3.1: Ohjelmiston kerrosarkkitehtuuri

3.3 Suunnittelumallit

Suunnittelumallit ovat erilaisia tekniikoita, jotka ovat olleet olennainen osa ohjelmistosuunnittelua 1990-luvun jälkeen [8]. Suunnittelumallit ovat hyväksi todettuja ratkaisuja tiettyihin ohjelmistotekniikassa toistuviin ongelmiin. Erilaisia suunnittelumalleja käyttämällä pyritään tukemaan arkkitehtuurisuunnittelua sekä yksityiskohtaisempaa suunnittelua. Suunnittelumalleista voidaan käyttää myös termiä arkkitehtuurimalli tai arkkitehtuurityyli, kun sillä kuvataan järjestelmää kaikkein korkeimmalla abstraktio tasolla. Eräs merkittävimmistä suunnittelumalleja kuvaavista teoksista on nk. Gang Of Four:n vuonna 1995 julkaisema kirja, joka sisältää useita yleisiä suunnittelumalleja [6].

3.3.1 Tarkkailija-suunnittelumalli

Arkkitehtuuri kuvaa siis komponenttien väliset rajapinnat ja komponenttien välisen vuorovaikutuksen. Komponenttien välistä suhdetta pyritään heikentämään Tarkkailija-suunnittelumallilla, jotta voitaisiin suunnitella helposti uudelleenkäytettäviä ja ylläpidettäviä komponentteja. [8]



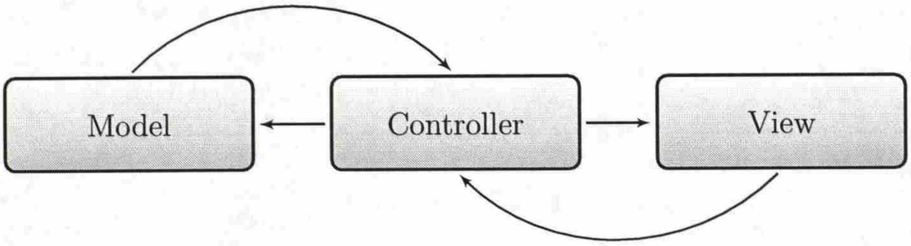
Kuva 3.2: Tarkkailija-suunnittelumalli

Mallissa jotakin palvelua tarjoava komponentti on lähde ja palvelua käyttävä komponentti on tarkkailija. Tarkkailija rekisteröityy lähteelle saadakseen tietoja, kun palvelu on saatavilla. Palvelu on tapahtuma (event), joka edustaa jotakin osaa sovellusdatasta. Kun palvelu on saatavilla, lähde käyttää takaisinkutsua välittääkseen tapahtuman tarkkailijalle, joka siten prosessoi tämän tapahtuman. Tarkkailija-suunnittelumallia käytetään yleisesti käyttöliittymien toteutuksessa. Mallin käyttäminen luvussa 3.3.2 kuvatun suunnittelumallin toteutuksessa saattaa vähentää järjestelmän kuormitusta takaisinkutsujen ansiosta, koska tarkkailijan ei tarvitse jatkuvasti kysyä lähteeltä uusista tapahtumista (polling) vaan lähde ilmoittaa, kun palvelu on saatavilla. [8]

Kuvassa 3.2 on esitetty Tarkkailija-suunnittelumallin luokkakaavio. *ConcreteObserver*-luokan olio rekisteröityy *Source*-oliolle saadakseen ilmoituksen tapahtumasta, eli esimerkiksi, kun uutta dataa anturilta on saatavana. Source-olio ilmoittaa kaikille rekisteröityneille olioille *Observer*-rajapinnan kautta uudesta tapahtumasta kutsumalla *processData()*-metodia kaikille rekisteröityneille olioille.

3.3.2 Malli-näkymä-ohjain

Ohjelmistolle on tärkeää käyttöliittymän ja sovelluslogiikan sekä datan erottaminen toisistaan, jotta käyttöliittymä voidaan toteuttaa erilaiseksi esimerkiksi työpöytä- ja mobiiliympäristöille ilman, että sovelluslogiikkaan tarvitsee tehdä muutoksia. Tätä ratkaisua tukeva sovelluskohtainen arkkitehtuurimalli on malli-näkymä-ohjain (Model-View-Controller, MVC), joka on esitetty kuvassa 3.3. Malli-näkymä-ohjain-arkkitehtuurimalli perustuu järjestelmän jakamiseen kolmentyyppisiin osiin. Malli edustaa ohjelmiston dataa, näkymä edustaa käyttöliittymässä olevia komponentteja, jotka näyttävät datan ja ohjain huolehtii siitä, että malli ja näkymä vastaavat toisiaan. MVC tai jokin sen muunnelma onkin yleisesti käytetty erityisesti käyttöliittymien komponenteissa. Periaatteessa pelkästään näkymä osaa muuttamalla saadaan aikaiseksi erilainen käyttöliittymän ulkoasu erilaisille alustoille. Arkkitehtuurimallin toteutuksessa käytetään yleisesti hyväksi luvussa 3.3.1 esiteltyä Tarkkailija-suunnittelumallia.



Kuva 3.3: Malli-näkymä-ohjain

Arkkitehtuurimallin yleisiä ongelmia ovat luokkien lukumäärän lisääntyminen eli järjestelmän monimutkaistuminen sekä mallin päivityskutsujen aiheuttama järjestelmän ylimääräinen kuormitus. Datan tai käyttöliittymän päivittyessä, ohjaimen täytyy huolehtia siitä, että malli ja näkymä vastaavat toisiaan, mikä vaati erilaisia päivityskutsuja. Koska näkymä- ja ohjainluokkia on yleensä vaikeaa käyttää toisistaan erillään muissa yhteyksissä, tässä ohjelmistossa nämä luokat yhdistetään. Näin voidaan luokkien lukumäärän vähentämisen ohella helpottaa myös järjestelmän kuormitusta, koska osa näkymän ja ohjaimen välisistä päivityskutsuista voidaan jättää tekemättä. [8]

3.3.3 Asiakas-palvelin-arkkitehtuuri

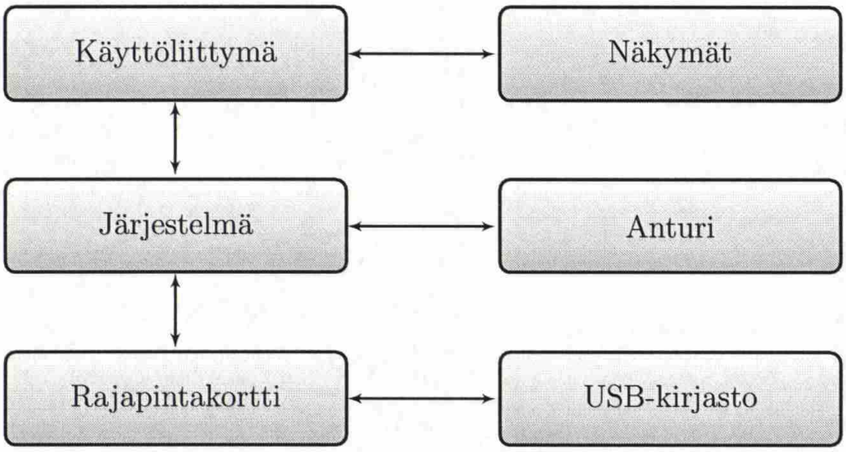
Coulouris [3] kuvaa hajautetun järjestelmän järjestelmäksi, jossa tietokoneet tai prosessit kommunikoivat keskenään vain lähettämällä viestejä. Anturi, rajapintakortti, tietokone ja ohjelmisto muodostavat tällaisen hajautetun järjestelmän. Ohjelmisto ja rajapintakortti kommunikoivat lähettämällä viestejä toisilleen makrokielellä ja anturi ja rajapintakortti kommunikoivat joko I^2C- tai SPI-väylän avulla. Järjestelmän hajauttamisesta on hyötyä etenkin tilanteissa, joissa käyttöliittymä halutaan siirtää kauemmaksi mitattavasta anturista, eli anturia halutaan lukea esimerkiksi lähiverkon yli.

Koska ohjelmiston tulee tarjota etärajapinta anturilta luettavaan dataan, tarvitaan jokin menetelmä prosessien väliseen kommunikointiin (Inter-Process Communication, IPC). Asiakas-palvelin-arkkitehtuuri on palveluperustainen arkkitehtuurityyli, joka kuvaa prosessien välisiä suhteita. Kahdelle tai useammalle prosessille annetaan selkeät roolit, joissa yksi prosessi toimii palvelimen roolissa ja muut asiakkaina. Roolit voivat kuitenkin vaihdella siten, että palvelin voi olla myös asiakas jollekin toiselle palvelimelle. Palvelimen tarjoama palvelu perustuu jonkin resurssin jakamiseen asiakkaiden kesken. Asiakkailla on palvelimen kautta mahdollisuus käsitellä resurssia tietämättä teknisiä yksityiskohtia resurssin käytöstä. [8]

Tässä työssä sovelletaan kaikkein perinteisintä mallia, jossa on yksi palvelin ja useita asiakkaita. Palvelimen hallinnoima ja asiakkaille jakama resurssi on anturilta kerättävä data. Jotta kaksi prosessia voi kommunikoida keskenään, täytyy prosessien välille muodostaa yhteys sekä kommunikaatiolle kehittää protokolla, jonka molemmat ymmärtävät. Protokollan suunnittelemiselta voidaan välttyä käyttämällä etämetodikutsua (Remote Method Invocation, RMI). Etämetodikutsu mahdollistaa kahden eri prosessissa olevan olion kommunikoinnin metodikutsujen avulla. Etämetodikutsu voidaan saada näyttämään melkein täysin läpinäkyvältä kutsujalle, eli olio ei välttämättä edes tiedä, että onko kutsuttava olion toisessa prosessissa tai kokonaan toisessa koneessa. Näin olio voi kutsua toisen olion metodeja eikä kommunikointiprotokollaa tarvitse erikseen suunnitella. Suunnitelmana on laajentaa tapahtumapohjainen toteutus toimimaan myös etämetodikutsujen kanssa.

Olio, jonka metodeja voidaan kutsua toisista prosesseista, on etäolio (Remote object) ja se toteuttaa etärajapinnan (Remote Interface). Näitä olioita voidaan kutsua etäolioreferenssin (Remote Object Reference) avulla. Kutsuun liittyvät tekniset yksityiskohdat, kuten parametrien ja paluuarvon sarjallistaminen tai kommunikointi kysely-vastaus-protokollalla, ovat tavallaan piilotettu ohjelmoijalta. Esimerkiksi, kun kutsutaan etämetodia Java RMI:n avulla, olio saa joko vastauksen, jos kutsu onnistui tai poikkeuksen virhetilanteen esiintyessä. Ohjelmoijan ei tarvitse tietää alemman ohjelmistokerroksen toteutuksesta. Java-ohjelmointikieltä käytettäessä hänen tulee vain huomioida se, että olioiden tulee olla sarjallistettavia (Serializable), jotta niitä voidaan käyttää RMI:n kanssa. Sarjallistettava olio voidaan lähettää kutsun parametrina tai saada paluuarvona. Sarjallistettava olio toteuttaa Java:ssa Serializable-rajapinnan. Olio sisältää olion luokan nimen ja version sekä muutamia muita tietoja. Näitä tietoja tarvitaan, jotta etäolio voi ladata tarvittaessa oikeat luokat etämetodikutsua suoritettaessa. [3]

3.4 Järjestelmän suunnittelu



Kuva 3.4: Suunniteltavan järjestelmän yleiskuva

Seuraavaksi kuvataan suunniteltavaa järjestelmää hahmottamalla ensin ongelman kohdealueen sanastoa vaatimusten pohjalta. Sanastoon kerätään ongelman kohdealueen keskeiset termit ja tarvittaessa tarkennetaan niiden merkityksiä tai suhteita. Sanaston on tarkoitus tukea ohjelmiston luokkakaavion hahmottelemista. Sanaston muodostamisen jälkeen hahmotellaan käyttötapauskaavio UML-kuvaustekniikan (Unified Modeling Language) avulla ja kuvataan muutamia käyttötapauksia yksityiskohtaisemmin. Tämän jälkeen esitetään ohjelmiston alustava luokkamalli. Järjestelmän suunnitelman tulisi ottaa huomioon asetetut tavoitteet ja tarjota menetelmät niiden saavuttamiseksi. Edellä mainittuja malleja käyttämällä saadaan aikaan hyvin laajennettava tapahtumapohjainen järjestelmä. Kuvassa 3.4 nähdään suunniteltavan järjestelmän yleiskuva, joka on muodostettu kuvan 1.1 pohjalta.

3.4.1 Käsiteanalyysi

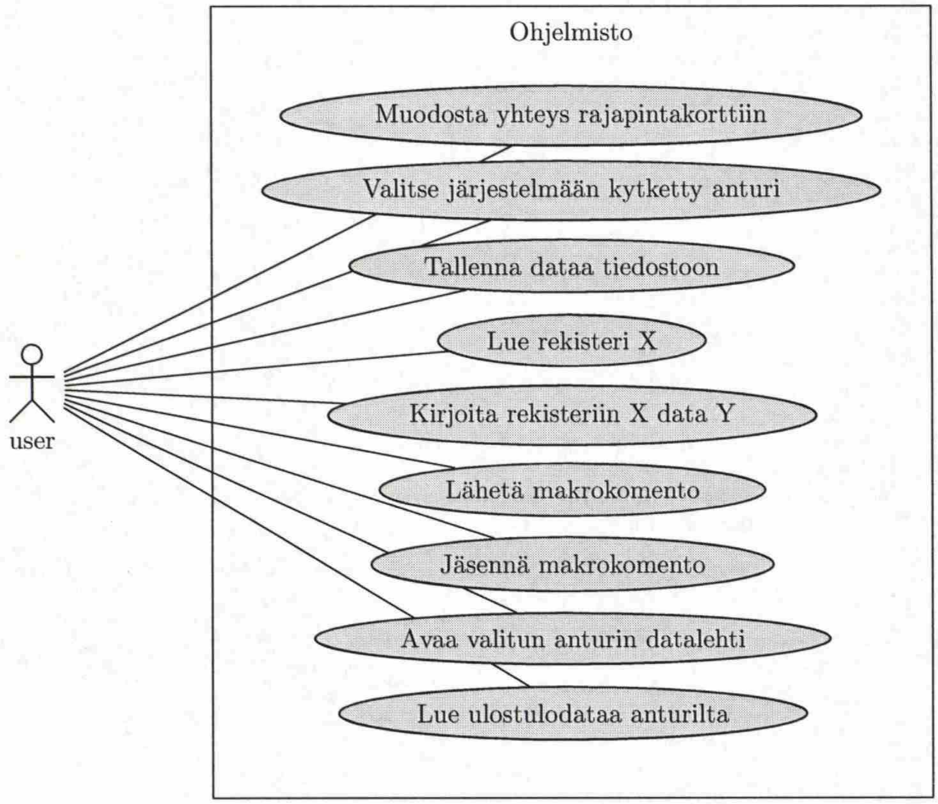
Taulukko 3.1 sisältää ohjelmiston ongelman kohdealueen keskeisimmät termit ja niiden selitykset. Näitä käsitteitä käytetään hyväksi ensimmäistä luokkamallia suunniteltaessa.

Taulukko 3.1: Ongelman kohdealueen termistö

Järjestelmä	Sovelluslogiikan kokonaisuus.
Käyttöliittymä	Kokonaisuus, joka sisältää erilaisia näkymiä.
Näkymä	Näyttää tietyn osan sovelluksen datasta.
Käyttäjä	Ohjelmiston käyttäjä.
Rajapintakortti	Elektroniikkakortti, johon anturi kiinnitetään.
Yhteys	Rajapintakortti käyttää yhteyttä komentojen lähettämiseen ja vastaanottamiseen.
Makrokomento	Rajapintakortille lähetettävä komento.
Jäsentäjä	Eräänlainen kääntäjä, jolla voidaan tarkistaa makrokomennon oikeellisuus.
Anturi	Rajapintakorttiin kiinnitettävä anturi.
Anturityyppi	Tietty anturiperheen tyyppi, jolla on tietyt ominaisuudet.
Ulostulodata	Anturilta kerättävä kiihtyvyysinformaatio / data.
Rekisteri	Anturin rekisteri, joka voidaan lukea tai mihin voidaan kirjoittaa dataa.
Etärajapinta	Rajapinta ulkoisille prosesseille anturilta saatavan datan lukemiseksi.

3.4.2 Käyttötapaukset

Järjestelmän käyttäjä (actor) on asiakas eli ihminen. Käyttötapauskuvaus kuvaa jonkin operaation, joka alkaa tietystä lähtökohdasta ja päättyy onnistuessaan tiettyyn tavoitteeseen. Kuvaukset voivat kuvata myös toiminnan poikkeavassa tilanteessa. Kuvassa 3.5 on esitetty muutamia käyttötapauksia esimerkkeinä sanaston pohjalta. Koska kuva pelkästään ei yleensä ole tarvittavan informatiivinen, esitetään myös vastaavat sanalliset selitykset muutamalle käyttötapaukselle.



Kuva 3.5: Käyttötapauskaavio

Käyttötapaus: Muodosta yhteys rajapintakorttiin

Tavoite:

Ohjelma listaa löytyneet rajapintakortit käyttäjälle. Käyttäjä määrittelee rajapintakortin tyypin ja muodostaa yhteyden rajapintakortin kanssa kommunikoimiseksi.

Käyttötapauksen kulku:

1. Käyttäjä valitsee rajapintakortin tyypin
2. Käyttäjä valitsee kortin ID:n listalta
3. Järjestelmä muodostaa yhteyden rajapintakorttiin

Poikkeuksellinen toiminta:

Yhteyttä ei muodosteta, jos informaatiotekstin palauttama rajapintakortin tyyppi ei vastaa valittua tyyppiä tai yhteyttä muodostaessa esiintyi jokin muu virhe. Käyttäjälle näytetään virheilmoitus.

Käyttötapaus: Kirjoita rekisteriin X data Y

Tavoite:

Käyttäjä valitsee kirjoitettavan rekisterin X sekä kirjoittaa datan Y sille varattuun kontrolliin ja painaa "Kirjoita"-kontrollia käyttöliittymässä, jolloin järjestelmä muodostaa, jäsentää ja lähettää komennon rekisterin kirjoittamiseksi.

Käyttötapauksen kulku:

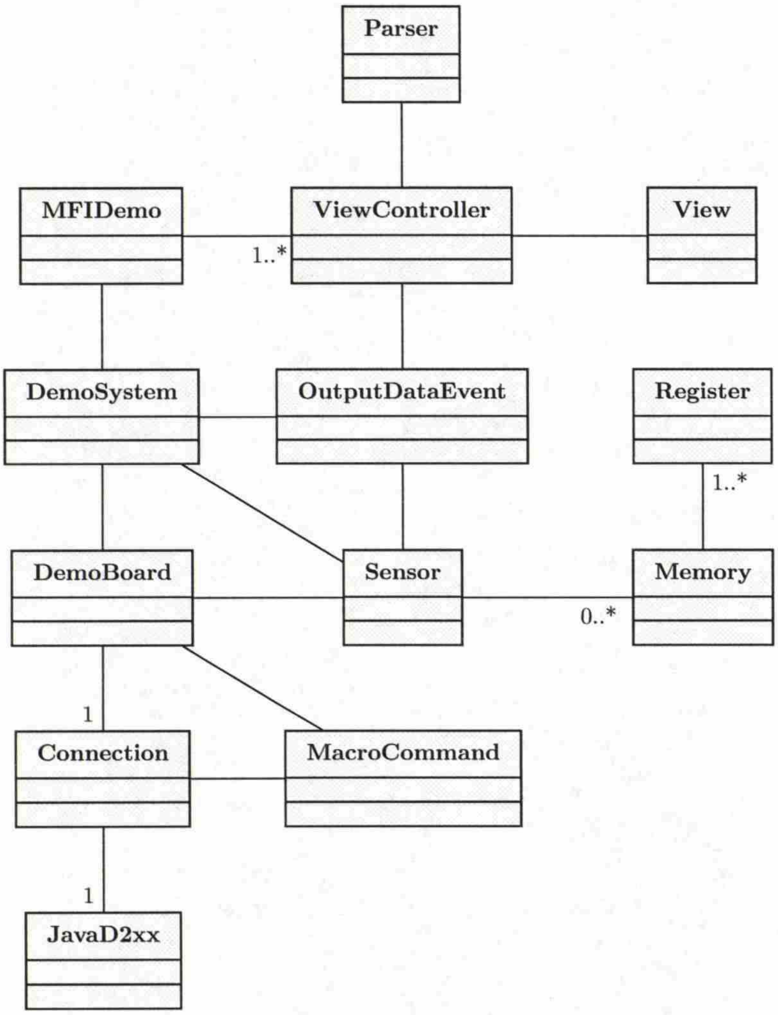
1. Käyttäjä valitsee kirjoitettavan rekisterin X
2. Käyttäjä syöttää kirjoitettavan datan Y
3. Järjestelmä muodostaa makrokomennon rekisterin kirjoittamiseksi
4. Järjestelmä tarkistaa makrokomennon jäsentäjän avulla
5. Järjestelmä lähettää makrokomennon rajapintakortille

Poikkeuksellinen toiminta:

Komentoa ei muodosteta, jos data sisältää virheitä tai yhteys katkennut. Käyttäjälle näytetään virheistä virheilmoitus.

3.4.3 Luokkamalli

Tutkimalla vaatimusmäärittelyn kuvauksia, muodostettua sanastoa ja käyttötapauskuvauksia voidaan muodostaa käsitteitä, jotka ovat komponentti- eli luokkaehdokkaita toteutettavaan ohjelmistoon. Kuvassa 3.6 on esitetty yksinkertainen luokkamalli, jossa on mainittu ohjelmiston kannalta tärkeimmät luokat. Luokkia muodostettaessa voidaan miettiä, että mitä luokkien olioiden tulee tietää ja mitä ne osaavat tehdä. Luokkien attribuutteja tai metodeja ei ole listattu tässä esityksessä ja suurin osa yhteyksien välisistä rajoitteista on jätetty pois. Suurin osa luokkien välisistä yhteyksistä on yhden suhde yhteen yhteyksiä. Seuraavassa on esitetty lyhyet kuvaukset näistä luokista.



Kuva 3.6: Luokkamalli

MFIDemo

MFIDemo-luokka sisältää ohjelman main()-metodin ja sen tehtävänä on alustaa käyttöliittymä ja muut tarpeelliset ominaisuudet, jotta käyttäjä voi valita rajapintakortin tyypin ja muodostaa yhteyden sekä aloittaa datan keräämisen anturityypin valitsemisen jälkeen. Tämän luokan tehtävänä on myös toteuttaa rajapinta, jonka avulla sovelluslogiikka voi lähettää viestejä käyttöliittymälle, esimerkiksi ilmoittaakseen virheistä. Luokan tulee luoda ainakin yksi näkymä eli *ViewController*-luokan olio.

ViewController

Näkymäohjain-luokka kokoaa datan visualisoimiseen tarvittavat oliot, kuten näkymän (*View*) ja osaa prosessoida anturilta saatavaa dataa tapahtumina (kts. 4.1.4). Näkymäohjain rekisteröityy *DemoSystem*-luokalle saadakseen tapahtumia, kun anturilta vastaanotetaan dataa. Kukin näkymäohjain-luokka periytetään ViewController-luokasta ja se voi esittää datan haluamassaan muodossa. Käyttöliittymä voi sisältää useita näkymiä.

DemoSystem

Luokka tarjoaa sovelluslogiikan tarjoamat palvelut muiden luokkien avulla. Luokasta periytetään anturikohtaiset järjestelmät, joiden tehtävänä on tarjota kyseiselle anturille ominaiset operaatiot. Näkymäohjain-luokat rekisteröityvät tälle oliolle vastaanottaakseen tapahtumia niiden esiintyessä. Luokka tarjoaa myös etärajapinnan, jonka avulla toiset prosessit voivat rekisteröityä vastaanottamaan tapahtumia. Etärajapinta ei kuitenkaan mahdollista sovelluslogiikan ohjaamista.

Sensor

Sensor-luokan tehtävänä on tarjota makrot kiihtyvyysdatan lukemiseksi sekä operaatiot kyseisen datan käsittelemiseksi. Luokan tulee tuntea anturin ominaisuudet sekä hallinnoida näitä ominaisuuksia. Ominaisuudet luetaan tiedostosta anturin tyypin perusteella. Luokan oliolta voidaan kysyä makrot rekistereiden lukemiseksi tai kirjoittamiseksi, anturin sarjanumero sekä muita anturiin liittyviä tietoja, esimerkiksi lista anturin sisältämistä muisteista tai rekistereistä.

OutputDataEvent

Kun ulostulodataa vastaanotetaan rajapintakortilta, *DemoSystem*-olio välittää tämän datan *Sensor*-oliolle, joka luo jäsentämästään datasta *OutputDataEvent*-olion. Tämä olio sisältää tapahtuman tiedot ja kiihtyvyysdatan. Olio palautetaan *DemoSystem*-oliolle, joka välittää sen kaikille rekisteröityneille kuuntelijoille.

DemoBoard

Luokan tehtävänä on tarjota operaatiot rajapintakortin kanssa kommunikoimiseksi. Luokan olio tietää, millaisia komentoja kyseiselle rajapintakortille voi lähettää ja mitä muita toiminnallisuuksia rajapintakortti tukee. Luokan olio käyttää *Connection*-luokkaa datan lähettämiseen ja vastaanottamiseen. Luokasta voidaan periyttää aliluokkia erilaisia rajapintakortteja varten, jotka voivat sisältää vain niihin liittyviä operaatioita.

Connection

Luokan olio ei tunne lähetettävää dataa, mutta se osaa lähettää ja vastaanottaa dataa rajapintakortilta käyttäen tiettyä yhteystyyppiä. Luokka tarjoaa operaatiot yhteyden hallinnoimiseksi ja esimerkiksi löydettyjen laitteiden listaamiseksi. Luokka käyttää *JavaD2xx*-luokan oliota erilaisten operaatioiden toteuttamiseksi.

MacroCommand

Luokan avulla voidaan muodostaa makrokomentoja merkkijonoina sekä jäsentää tietoa saaduista vastauksista.

JavaD2xx

Koska FTDI:n ajurikirjaston funktioiden kutsuminen suoraan Java:n luokkakirjaston kautta ei ole mahdollista, tarvitaan kääreluokka, jonka avulla voidaan kutsua näitä funktioita (kts. 4.4). Luokka tarjoaa siis erilaiset operaatiot FTDI:n USB-laitteelle. Luokan avulla voidaan kommunikoida FTDI:n ajurikirjaston kanssa. Kirjasto vaatii Java toteutuksen lisäksi natiivi toteutuksen C-kielellä, joka joudutaan kääntämään erikseen eri käyttöjärjestelmille.

Memory

Anturiin voi liittyä useita erilaisia muisteja. Tämän luokka sisältää attribuutin muistin kuvaukselle sekä listalle rekistereitä, jotka kuuluvat tähän muistiin (kts. 4.2.1).

Register

Luokka kuvaa anturin rekisterin attribuutit ja erilaiset operaatiot niiden käsittelemiseksi (kts. 4.2.1).

Parser

Luokka osaa jäsentää makrokomentoja, eli sillä voidaan tarkastaa makrokomento ennen komennon lähettämistä rajapintakortille (kts. 4.3).

Luku 4

Ohjelmiston toteutus

Tässä luvussa kuvataan toteutettua ohjelmistoa yleisesti sekä toteutukseen liittyviä yksityiskohtia, kuten erilaisten antureiden määrittelemistä, makrokomennoille suunniteltua makrokielen jäsentäjää sekä USB-kommunikointia varten kehitettyä *JavaD2xx*-kirjastoa.

4.1 Yleiskuvaus

Kehitysympäristöksi valittiin jo käytössä oleva Eclipse IDE (Integrated Development Environment). Eclipse-kehitysympäristössä on monia hyviä ominaisuuksia, kuten luokan kenttien ja metodien listaus, syntaksin korostus, koodiassistentti sekä mahdollisuus generoida *javadoc*-dokumentaatio luokille suoraan valikoista. Eclipse:n perusversio ei sisällä graafista editoria käyttöliittymien suunnitteluun, kuten esimerkiksi *NetBeans IDE*. Mutta koska käyttöliittymät rakennetaan dynaamisesti anturin ominaisuuksien mukaan, graafisen editorin puutteesta ei ole haittaa. Ohjelmiston lopullinen jakelupaketti käännetään *Apache Ant*-työkalulla, joka on tarkoitettu erityisesti Java-sovellusten kääntämiseen, paketoimiseen, testaamiseen ja suoritukseen. Ohjelmisto käännetään Ant:n käyttämien tehtävien avulla, jotka annetaan työkalulle *build.xml*-tiedoston avulla. Tehtävät kääntävät Java-luokat ja paketoivat sen jälkeen ohjelmiston ZIP-tiedostoon. Paketoidun tiedoston purkamalla asiakas voi käynnistää ohjelman skriptitiedostosta. Ohjelman käynnistymiseen liittyy erilaisia toimenpiteitä käyttöjärjestelmästä riippuen. Kaikissa käyttöjärjestelmissä täytyy Java:n virtuaalikoneelle (Java Virtual Machine, JVM) määritellä, mistä käännetyt Java-luokat löytyvät. Tämä tapahtuu määrittelemällä polut CLASSPATH-ympäristömuuttujaan.

Linux-käyttöjärjestelmässä täytyy myös ftdi_sio- ja usbserial-sarjaporttiajurimoduulit poistaa käytöstä *rmmod*-työkalulla, jotta FTDI:n laitetta voidaan käyttää USB-laitteena. Lisäksi USB-laitteen oikeuksiin täytyy tehdä muutoksia. Tämä tehdään muodostamalla sääntö FTDI:n USB-laitteelle *udev*-moduulin avulla.

Ohjelman parametrit, käyttöliittymän tekstit ja antureiden ominaisuudet määritellään tiedostoissa, jotka sisältävät avain–arvo-pareja (kts. 4.1.1). Java-kielen luokkakirjasto sisältää metodit näiden tiedostojen lukemiseksi ja tallentamiseksi sekä arvon hakemiseksi tai asettamiseksi avaimen perusteella. Ohjelman parametrit ja anturin ominaisuudet ladataan *Properties*-luokan avulla. Luokka sisältää metodin *load(InputStream)*, jolle annetaan parametrina tiedostovirta, joka on muodostettu halutun tiedoston mukaan. Tämän jälkeen avaimen arvo voidaan hakea *getProperty(String)*-metodilla, joka palauttaa arvon merkkijonona, joka voidaan sitten muuttaa sopivaksi tietotyypiksi.

Käyttöliittymän kielitiedostojen rakenne on samanlainen edellä mainittujen tiedostojen kanssa, mutta sitä käytetään *ResourceBundle*-luokan avulla. Luokka sisältää lokaalikohtaiset oliot. Ohjelman parametrit sisältävät tiedot *Locale*-olion muodostamista varten. Ohjelma luo näiden tietojen avulla määritellylle kielelle Locale-olion, joka annetaan parametrina *ResourceBundle.getBundle(getClassName(), locale)*-metodille. Tämä luo ResourceBundle-olion, jolta voidaan kysyä lokalisoitu teksti käyttöliittymään avaimen avulla käyttäen *getString(String)*-metodia. Tämäkin metodi palauttaa avaimen arvon merkkijonona, joka voidaan näyttää käyttöliittymässä.

4.1.1 Hakemistorakenne

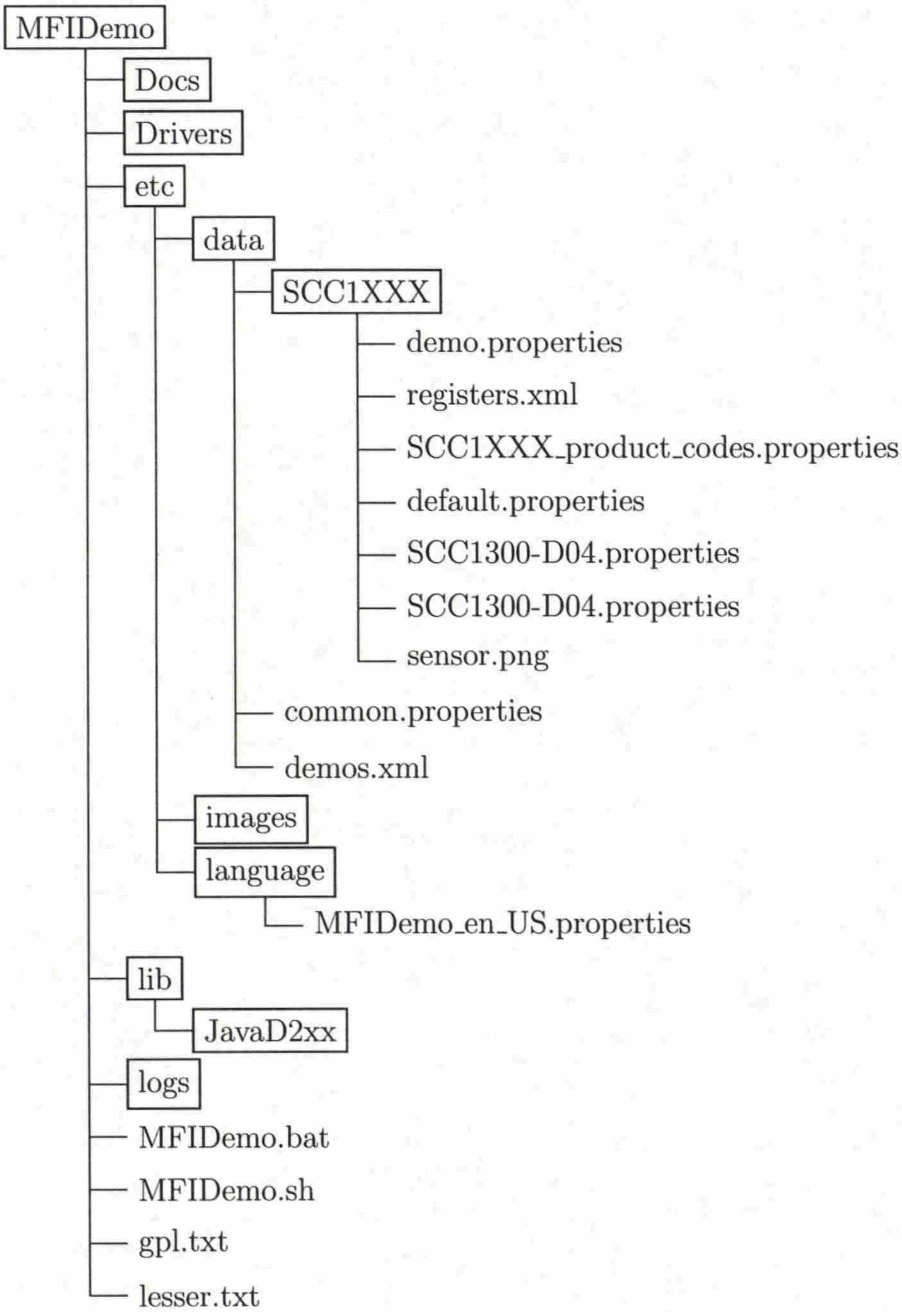
Ohjelmiston pelkistetty hakemistorakenne on esitetty kuvassa 4.1. Kuvaan on merkitty SCC1300-anturin tarvitsemat tiedostot ja kansiot. Hakemistorakenne pyrkii luomaan laajennettavan ja helposti ymmärrettävän kokonaisuuden. Juurihakemisto sisältää skriptitiedostot ohjelmiston käynnistämiseksi, erilaisia alihakemistoja sekä GPL- (GNU General Public License) ja LGPL-lisenssitiedostot (GNU Lesser General Public License). *Docs*-hakemisto sisältää ohjelmiston olennaiset dokumentit, esimerkiksi käyttöohjeen. *Drivers*-hakemisto sisältää ohjelmiston kanssa testatut FTDI:n laiteajurit. *lib*-hakemisto sisältää useita kirjastoja ryhmiteltyinä alihakemistoihin, jotka sisältävät tarvittavat natiivi- ja Java-kirjastot. Java-kirjastot on paketoitu JAR-tiedostoihin (Java Archive). Näiden JAR-tiedostojen polku asetetaan CLASSPATH-ympäristömuuttujaan ohjelman käynnistävässä skriptitiedostossa, jotta Java:n virtuaalikone löytää luokka-tiedostot. *logs*-hakemistoon tallennetaan ohjelman suorituksen aikana kirjoitetut loki-tiedostot, yksi tiedosto jokaista suorituskertaa kohden. Tiedostot nimetään seuraavan kaavan mukaisesti: pp_kk_vvvv_tt_mm_ss.log.

etc-hakemisto sisältää ohjelman yleiset asetukset sekä *data*- ja *language*-alihakemistot. *etc/language*-hakemisto on tarkoitettu tiedostoille, jotka sisältävät käyttöliittymän tekstit eri kielille. MFIDemo_en_US.properties-tiedosto sisältää tekstit englannin kielelle. Tiedoston nimessä on luokan nimen jälkeen kahdella pienellä kirjaimella kielen tunnus (ISO-639) ja kahdella isolla kirjaimella maakoodi (ISO-3166). Luokka, jonka nimi on tiedostonimessä, on se luokka, jossa ohjelman *main()*-metodi on. Ohjelman käyttöliittymän kielen voi vaihtaa luomalla uuden tiedoston kyseiselle kielelle tähän kansioon ja määrittelemällä sen käyttöön *etc/data/common.properties*-tiedostossa. *etc/-data*-hakemisto sisältää ohjelmiston oleellisimmat alihakemistot ja tiedostot. *demos.xml*-tiedosto määrittelee ohjelman tukemat anturiperheet (kts. 4.2). Lisäksi jokaiselle anturiperheelle tulee olla oma alihakemistonsa, joissa on parametritiedostot eri anturityypeille, tiedosto ID-koodeille, parametritiedosto anturiperheen yleisille asetuksille ja tiedosto anturin rekistereiden kuvauksille.

4.1.2 Käyttöliittymätekniikat

Käyttöliittymän toteutuksessa käytettiin pääsääntöisesti Java-kielen Swing-komponentteja. Ohjelmistoon toteutettiin myös muutamia omia komponentteja luomalla aliluokkia *JComponent*-luokasta ja kuormittamalla sen *paintComponent(Graphics)*-metodia. Ohjelmisto käyttää erilaisten kuvaajien piirtämiseen kolmannen osapuolen avoimen lähdekoodin jChart2D-kirjastoa. Kirjasto on julkaistu LGPL-lisenssillä.

Swing-komponentit on toteutettu Java-kielellä, joten ne eivät ole sidoksissa käyttöjärjestelmän arkkitehtuuriin. Swing-kirjasto tarjoaa tavan näiden komponenttien ulkoasun ja toiminnallisuuden muokkaamiseksi. Ohjelma asettaa ulkoasutyylin (look-and-feel) ohjelman *main()*-metodissa. Asetettava tyyli saadaan *UIManager.getSystemLookAndFeelClassName()*-metodilta. Tyyli muuttaa komponenttien ulkonäön ja toiminnallisuuden käyttöjärjestelmän mukaan. Kun ohjelmaa ajetaan esimerkiksi Windows-käyttöjärjestelmässä, komponentit saadaan näyttämään samalta, kuin muut Windows-käyttöjärjestelmän komponentit. Java FX -kirjastoa ei käytetty tässä ohjelmistossa käyttöliittymän rakentamiseen, koska haluttiin säilyttää mahdollisuus käyttää ohjelmistoa vanhemman Java SE 6 version kanssa. Käytännössä on huo-



Kuva 4.1: Ohjelmiston hakemistorakenne

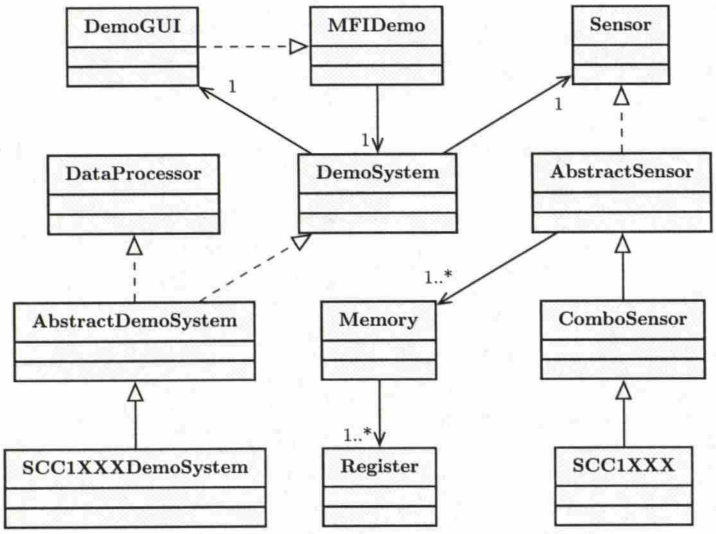
mattu, että monella asiakkaalla on vielä tämän työn tekemisen aikana Java SE 6 käytössä. Java FX v2.2, joka sisältää myös Linux-tuen, vaatii vähintään Java SE 7 Update 6 version.

4.1.3 Sovelluslogiikka

MFIDemo-luokka käyttää *DemoSystem*-rajapintaa sovelluslogiikan kanssa kommunikoimiseksi. Rajapinta sisältää metodit, joilla näkymät voivat rekisteröidä itsensä kuuntelemaan tapahtumia sekä metodit, joilla voidaan lukea tai kirjoittaa anturin rekistereitä sekä muita metodeja. Monet, kaikille järjestelmille yhteiset, metodit on toteutettu *AbstractDemoSystem*-luokassa. Tällaisia metodeja ovat esimerkiksi sovelluslogiikan ohjaukseen tarkoitetut metodit (mm. *setCalculateFFT()* tai *restartDataCollection()*) sekä tilojen kysymiseen tarkoitetut metodit (mm. *isCollectingData()* tai *isStreamingData()*).

AbstractDemoSystem-luokkaa laajennetaan luokilla, jotka sisältävät anturiperheeseen liittyviä toimintoja. Kuvasta 4.2 nähdään, miten periytymistä käytetään SCC1300-anturin toiminnallisuuksien toteuttamiseen. Esimerkiksi, kun käyttäjä valitsee alkuvalikosta anturityypiksi SCC1300-D04, *MFIDemo*-luokka luo *SCC1XXXDemoSystem*-olion. Tämä olio tietää, miten sen tulee asettaa esimerkiksi tietyt toimintotilat kyseisen anturin kanssa. Näin voidaan *DemoSystem*-rajapinnan kautta kutsua *setOperationMode(String mode)*-metodia ja järjestelmän toiminnallisuus riippuu siten valitusta anturista. *SCC1XXXDemoSystem*-luokka sisältää myös metodeja muistitestien suorittamiseksi, joita ei voida suorittaa muille antureille. *DemoGUI*-rajapinta sisältää metodit, joiden avulla sovelluslogiikka lähettää viestejä käyttöliittymälle.

Kun ulostulodatan kerääminen käynnistetään, *AbstractDemoSystem*-olio luo säikeen, joka lukee ulostulodataa *Sensor*- ja *Connection*-luokkien avulla. *AbstractDemoSystem* toteuttaa *DataProcessor*-rajapinnan. Rajapinta määrittelee metodin *dataReceived(String data)*, jota kutsutaan heti, kun merkkejä on vastaanotettu ja metodin *outputDataFrameReceived(String frame)*, jota kutsutaan, kun tietty anturiin liittyvä merkkijono on vastaanotettu. *outputDataFrameReceived()*-metodin toteutuksen tehtävänä on välittää data *Sensor*-oliolle, joka jäsentää merkkijonosta esimerkiksi kiihtyvyysarvot ja tilabitit ja palauttaa *OutputDataEvent*-olion sisältäen tapahtuman tiedot. Tämä olio lähetetään sitten kaikille rekisteröityneille kuuntelijoille kutsumalla olioiden *processOutputData()*-metodia. *dataReceived(String data)*-metodilla voidaan lähettää dataa kuuntelijalle, joka haluaa kaiken datan heti, kun se on vastaanotettu. Näiden olioiden tulee rekisteröidä itsensä *addDataListener(DataListener l)*-metodilla. Tätä toiminnallisuutta käyttää esimerkiksi *MacroCommandViewController*-näkymäohjain näyttääkseen kaiken vastaanotetun datan rajapintakortilta.

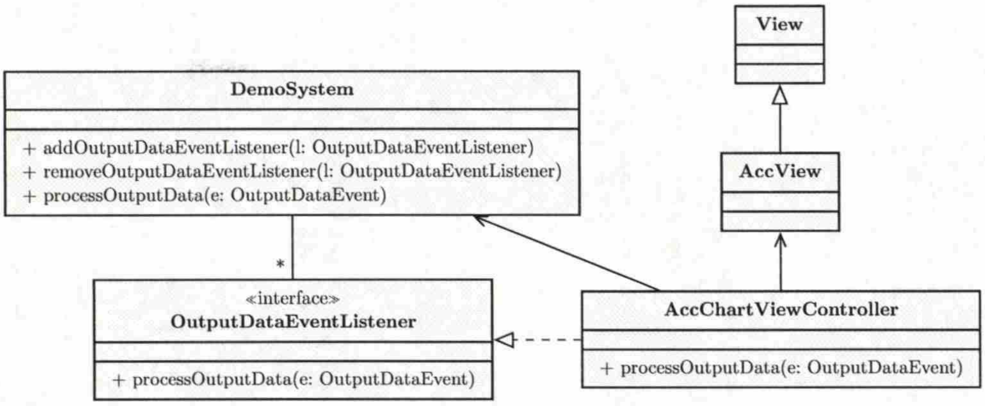


Kuva 4.2: Sovelluslogiikan kuvaus

4.1.4 Näkymät

MFIDemo-luokka periytyy *JFrame*-luokasta. Se muodostaa sovelluksen ikkunan, jossa erilaiset näkymät näytetään välilehdillä. Se sisältää myös ohjelman *main()*-metodin, jonka ohjelman käynnistävä säie (initial thread) suorittaa. Metodi asettaa ulkoasutyylin käyttöliittymälle ja luo *Runnable*-olion, jonka tehtävänä on muodostaa käyttöliittymä. Tämä olio suoritetaan EDT-säikeessä (Event Dispatch Thread). Tämän jälkeen ohjelma toimii tapahtumapohjaisesti.

Näkymät tarjoavat kontrollit sovelluslogiikan ohjaamiseen sekä vastaavat anturilta saatavan datan esittämisestä käyttäjälle. Erialiset näkymät lisätään käyttöliittymässä välilehtiin, jotka ovat käyttäjän valittavissa. Kun käyttöliittymää luodaan, muodostetaan *demos.xml*-tiedostossa määriteltyjen näkymien perusteella näkymäohjaimet (*ViewController*), jotka luovat myös näkymän (*View*). Tämä näkymä sijoitetaan välilehteen, minkä jälkeen kutsutaan näkymäohjaimen *startController(DemoSystem demoSystem)*-metodia. Näkymäohjaimen tulee tässä metodissa rekisteröidä itsensä vastaanottamaan tapahtumia sovelluslogiikalta.



Kuva 4.3: AccChartViewController

Kuvassa 4.3 nähdään esimerkkinä luokka *AccChartViewController*, johon on yhdistetty sekä näkymä, että ohjain. Luokan olio rekisteröityy *DemoSystem*-oliolle saadakseen ilmoituksen tapahtumasta, eli kun uutta dataa anturilta on saatavana. *DemoSystem*-olio ilmoittaa kaikille rekisteröityneille olioille *OutputDataEventListener*-rajapinnan kautta uudesta tapahtumasta kutsumalla rekisteröityneen olion *processOutputData()*-metodia. Rekisteröityneen olion tulisi vain tallentaa tapahtuma takaisinkutsun aikana ja prosessoida se myöhemmin. Tällä menetelmällä pyritään siihen, että järjestelmä ei hidastu, vaikka datan prosessointi käyttöliittymään kestäisikin kauemmin. Näkymän tulee käyttää Swing-kirjaston *Timer*-oliota prosessoidakseen takaisinkutsulla tallennettua dataa tietyn väliajoin.

Sovelluslogiikan ohjaamiseen suunnitellut näkymät eivät vastaanota dataa tapahtumien kautta. Ne odottavat vain käyttäjältä syötettä, minkä jälkeen ne lähettävät komentoja sovelluslogiikalle annetun syötteen mukaan.

4.1.5 Loki

Kun ohjelman suorituksessa tapahtuu jokin virhe, ohjelman tulisi tallentaa siitä tiedot loki-tiedostoon. Näitä tietoja tulisi voida käyttää virheen syiden selvittämiseksi myöhemmin, koska asiakkaan on yleensä vaikea kuvata, mitä suorituksen aikana tapahtui. Asiakkaan on helpompaa vain lähettää loki-tiedosto tutkittavaksi. Listaus 4.1 näyttää, miten *MFIDemo*-luokka alustaa *LOGGER*-muuttujan. *java.util.logging*-paketti sisältää *Logger*-luokan, jolla voidaan muodostaa loki-viestejä ja kirjoittaa niitä loki-tiedostoihin. *getLogger(String name)*-metodilla kysytään luokan nimen perusteella *Logger*-olio. Jokainen luokka voi muodostaa oman *Logger*-olion vastaavasti. Nämä oliot muodostavat Java-luokkien paketointia vastaavan hierarkian, joka tulostuu myös viesteihin, eli viestistä voidaan nähdä, mikä luokka sen on tulostanut ja mihin pakettiin kyseinen luokka kuuluu.

Listaus 4.1: LOGGER-olion luominen

```
private final static Logger LOGGER = Logger.getLogger(MFIDemo.class.getName());
```

Jotta jokaisen luokan ei tarvitse erikseen määritellä tiedostoa, johon lokiviestit kirjoitetaan, *MFIDemo*-luokka sisältää *createLogger()*-metodin. Listauksesta 4.2 nähdään, miten metodissa haetaan ensin yläluokan *Logger*-olio, jolle luodaan käsittelijä (*FileHandler*). Syvemmällä hierarkiassa olevat oliot käyttävät oletuksena yläluokan käsittelijöitä. Koska tässä tapauksessa yläluokan olio on *fi.murata.ds*-paketissa, joka on ylin taso ohjelman luokkahierarkiassa, kaikkien luokkien *Logger*-olioiden kirjoittamat loki-viestit tallennetaan samaan tiedostoon.

Listaus 4.2: LOGGER-olion alustus

```
String logFileName = "logs"+ File.separator + time + ".log";
txtLogFile = new FileHandler(logFileName, false);

txtLogFormatter = new SimpleFormatter();
txtLogFile.setFormatter(txtLogFormatter);

Logger logger = LOGGER.getParent();
logger.addHandler(txtLogFile);
logger.setLevel(Level.CONFIG);
```

4.2 Antureiden määrittely

Tässä kappaleessa kuvataan, miten anturit määritellään järjestelmään niin, että niitä on helppo lisätä myöhemmin ja että niitä voidaan käsitellä geneerisillä ohjelmistokomponenteilla. Käyttäjän tulee valita anturiperhe ohjelman alkuvalikossa. Tämän tiedon avulla näytetään käyttäjälle mahdolliset anturityypit, joista käyttäjän tulee valita oikea tyyppi. Osa anturiperheistä tukee myös anturityypin automaattista tunnistamista. Anturin tyyppi selvitetään lukemalla anturin ID-rekisterin arvo anturilta ja vertaamalla tätä arvoa etc/data/<path> polusta löytyvän <Anturi>_product_codes.properties-tiedoston arvoihin.

Ohjelmistossa käytettävät anturit ja niiden rekisterit määritellään käyttäen XML-merkintäkieltä (Extensible Markup Language). W3C (World Wide Web

Consortium) spesifikaatio [21] määrittelee XML:n merkintäkieleksi, joka kuvaa sekä tiedon, että sen loogisen rakenteen. XML on SGML-metakielen osajoukko. XML on suunniteltu helposti luettavaksi niin ihmiselle kuin koneellekin sekä tukemaan laajaa joukkoa erilaisia applikaatioita. XML-dokumentteja käytetään hyvin monissa erilaisissa yhteyksissä, esimerkiksi erilaisten tietorakenteiden kuvauksissa tai käyttöjärjestelmän konfigurointitiedostoina. Java FX FXML ja Android-sovellukset käyttävät XML-dokumentteja käyttöliittymän rakenteen kuvaamiseen irrallaan sovelluslogiikasta. JAXP-ohjelmointirajapinta (Java API for XML Processing) tarjoaa metodit XML-dokumenttien lukemiseksi ja kirjoittamiseksi. JAXP:a käytetään muodostamaan DOM (Document Object Model), joka on annetusta dokumentista jäsennetty puumainen muistissa oleva tietorakenne [20].

XML-dokumentti voi olla hyvin muodostettu, jolloin se täyttää kaikki spesifikaation asettamat rajoitukset tai validi, jolloin se täyttää myös määritellyn DTD:n (Document Type Definition) rajoitteet. XML-dokumentti alkaa prologilla, joka määrittelee XML-kielen version sekä mahdollisesti merkistön koodauksen ja muita tietoja. Listaus 4.3 määrittelee XML-kielen versioksi 1.0, UTF-8 merkistön sekä sen, että tämä XML-dokumentti on itsenäinen eli se ei viittaa muihin XML-dokumentteihin.

Listaus 4.3: Osa demos.xml tiedostoa

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<demos>
    <!-- Demo definitions -->
    <demo name="SCA103T">
        <!-- Sensor type -->
        <type>inclinometer</type>

        <!-- Demo description -->
        <description>
        Demo software for 3D-MEMS-based single axis inclinometer family
        </description>

        <!-- Relative path to sensor specific files -->
        <path>SCA103T</path>

        <!-- Views supported by this sensor type -->
        <views>0,9,7</views>
    </demo>
    ...
</demos>
```

XML-dokumentti koostuu vähintään yhdestä elementeistä. Jos elementtejä on useampia, vain yksi niistä voi olla pääelementti (root element). Elementti koostuu alkutagista (tag), sisällöstä ja lopputagista. Elementti kuvataan tageilla seuraavasti, sisältönä voi olla merkkijono, toinen elementti tai esimerkiksi CDATA lohko, jolloin sisältöä ei jäsennetä. Listaus 4.3 on osa *demos.xml*-tiedostoa, joka määrittelee ohjelmistossa käytettävät anturit. Pääelementti <demos> sisältää listan <demo>-elementtejä, joista kukin kuvaa yhden ohjelmistossa käytettävän anturiperheen. *name*-attribuutti määrittelee anturin nimen. Muut elementit määrittelevät anturin komponentin tyypin (<type>), kuvauksen (<description>), anturiin liittyvät näkymät (<views>) sekä suhteellisen polun (etc/data/<path>) anturin vaatimiin muihin tiedostoihin. <type>-elementtiä käytetään luotaessa käyttöliittymää, samassa näkymässä voidaan näyttää eri tietoja eri tyypeille. Elementti <views> kuvaa indeksit, jotka vastaavat *ViewTypes*-literaaliluokan (enum) näkymille määrittelemiä indeksejä. Nämä näkymät luodaan alkuvalikon jälkeen ohjelman käynnistyessä.

4.2.1 Rekisterit

Melkein jokainen digitaalisella liitännällä varustettu anturi sisältää erilaisia käyttäjän luettavissa ja kirjoitettavissa olevia rekistereitä. Rekistereiden osoitteet, leveydet ja nimet kuitenkin poikkeavat toisistaan eri anturien välillä. Tässä diplomityössä kehitettiin käyttöliittymään geneerinen komponentti, joka osaa näyttää erilaisten rekistereiden sisältöjä ja mahdollistaa näiden rekistereiden lukemisen ja kirjoittamisen. Jotta tällainen komponentti voitiin toteuttaa, rekistereiden ominaisuudet ja sisältö täytyi määritellä tarkasti kullekin anturille erikseen.

Koska XML-dokumentteja on helppo muodostaa ja lukea ohjelmallisesti, XML-kieltä käytettiin myös anturin rekistereiden määrittelyssä. Anturin rekisterit määritellään *registers.xml*-tiedostossa, jonka tulee löytyä *demos.xml*-tiedostossa määritellystä polusta */etc/data/<path>*. Listaus 4.4 on esimerkki rekistereiden määrittelystä SCC1300-anturille. Pääelementin sisältö on lista muisti-elementtejä (<memory>). Yksi tällainen elementti määrittelee kyseiseen muistiin liittyvät rekisterit (<register>). Jos esimerkiksi anturissa on kaksi ASIC-piiriä, kuten yllä olevassa esimerkissä, voidaan molemmille määritellä rekisterit erikseen näiden elementtien avulla. Jokainen rekisterielementti sisältää attribuutteina osoitteen, rekisterin leveyden sekä nimen. Tämän lisäksi rekisteriin voidaan liittää kuvaus ja n-kappaletta bitti-elementtejä (<bit>), jotka kuvaavat yksittäisiä bittejä tai joukkoja bittejä, joilla voi olla nimi, kuvaus ja oikeudet (access mode), joka määrittelee voiko bittiä lukea ja kirjoittaa. *RegisterParser*-luokan olio osaa jäsentää nämä kuvaukset ja luoda niistä listan *Memory*-luokan olioita, joista kukin sisältää listan *Register*-luokan olioita. Nämä listat tallennetaan *Sensor*-luokan oliolle, jolta ne

Listaus 4.4: Rekistereiden määrittely

```
<registers>
  <memory id="Accelerometer">
    <register address="0x00" width="8" name="Revision ID (REVID)">
      <description>
        Each product family component type has unique identification
        number, which is stored to non-volatile memory. This number
        can be used for example in production line to check that
        correct component is mounted to the system. In some cases it
        may be used for MCU software controlling.
      </description>
      <bit id="7:0">
        <name>REVID</name>
        <description>ASIC revision ID number</description>
        <accessMode>ro</accessMode>
      </bit>
    </register>
    ...
  </memory>
  <memory id="Gyroscope">
    ...
  </memory>
</registers>
```

voidaan kysyä tarvittaessa.

Määrittelemällä rekisterit edellä kuvatulla tavalla, käyttöliittymään voitiin suunnitella komponentti *RegisterSetupViewController*, joka käyttää tätä määrittelyä. Komponentti tarjoaa kontrollit muistin sekä rekisterin valitsemiseen, havainnollisen esityksen rekisterin sisällöstä rekisterin kuvauksen perusteella sekä mahdollisuuden muuttaa rekisterin sisältöä ja kontrollit rekisterin lukemiseen ja kirjoittamiseen.

4.2.2 Ominaisuudet

Tärkeimmät antureiden ominaisuudet määritellään joukkona (set) Javan *Properties* luokan avulla. Jokaiselle anturityypille tulee olla <anturin_tyyppi> .properties tiedosto, jossa anturityypin akselit, herkkyydet ja muut ominaisuudet määritellään avain–arvo-pareina. Java-kieli sisältää metodit näiden tiedostojen lukemiseksi ja tallentamiseksi sekä arvon hakemiseksi tai asettamiseksi avaimen perusteella. Tavoitteena on, että tiettyjä operaatioita varten parametrit voidaan hakea käyttämällä avainta, jonka arvo ladataan anturikohtaisesti näiden tiedostojen avulla. Näin ohjelmistossa voidaan käyttää geneerisiä komponentteja, jotka toimivat oikein eri antureiden ja anturityyppien kanssa. Listauksessa 4.5 on esimerkkinä avain–arvo-parit SCC1300-D04-anturityypille SCC1300-D04.properties tiedostosta. Listauksesta voidaan nähdä, että kyseessä on yhdistelmäanturi, jossa on 3-akselinen kiihtyvyysanturi ja x-akselin suuntainen kulmanopeusanturi. Listaus määrittelee myös toiminta-alueet, herkkyydet ja SPI-väylän maksimi kellotaajuuden.

Listaus 4.5: SCC1300-D04.properties

```
gyro.spi.freq.max=8000K
gyro.range.min=-300
gyro.range.max=300
gyro.sensitivity=18
gyro.axis=x,-,-
gyro.lsb.shift.count=2
acc.spi.freq.max=8000K
acc.range.min=-6
acc.range.max=6
acc.sensitivity=650
acc.axis=x,y,z
acc.lsb.shift.count=2
```

DemoSystem-rajapinta määrittelee *initializeSensor()*-metodin, jonka tulee huolehtia kaikista vaadittavista operaatioista tai alustuksista, joita anturille täytyy tehdä, ennen kuin kiihtyvyysarvoja voidaan lukea. Metodin tulee kutsua *Sensor*-rajapinnan sisältämää *readAsicParametersFromFile(String)*-metodia, jolla ladataan anturin ominaisuudet. Metodille annetaan parametrina polku yllä kuvattuun tiedostoon. Rajapinta sisältää myös *readAsicSpecificParametersFromFile(String)*-metodin, jolla voidaan ladata jonkun tietyn anturin ominaisuuksia esimerkiksi sarjanumeron perusteella. Näin voidaan määritellä erikseen esimerkiksi yhden anturin jokaiselle akselille mitattu herkkyys.

4.3 Makrokielen jäsentäjä

Eräs vaatimuksista oli toteuttaa toiminnallisuus makrokomentojen syntaksin tarkistamiseksi. Makrokieli voidaan nähdä yksinkertaisena ohjelmointikielenä, jolla voidaan kommunikoida rajapintakortin kanssa (kts. 2.4). Ohjelmointikielen syntaksi kuvataan kääntäjälle yleensä kontekstittomalla kieliopilla [1]. Kontekstiton kielioppi koostuu:

- päätesymboleista eli terminaaleista
- välikkeistä (yksi välike on määritelty alkusymboliksi)
- äärellisestä joukosta produktioita

Terminaalit määrittelevät kielen alkeellisimmat symbolit. Välike kuvaa joukon merkkijonoja, jotka on muodostettu terminaaleista. Produktioiden vasen puoli on välike ja oikea puoli on sarja välikkeitä ja / tai terminaaleja. Produktion vasen puoli, kuvaa tavallaan rakenteen nimen ja oikea puoli sen kirjoitetun ulkoasun. Ohjelmointikielen kääntäjän alkupään (front end) ensimmäiset komponentit ovat selaaja ja jäsentäjä. Selaaja puskuroi syötettä, tekee mahdollisia esikäsittelyjä (kommenttien poisto) sekä syntaktisen tarkastuksen. Tämän jälkeen se välittää syötteen jäsentäjälle. Jäsentäjä käyttää kielioppia hyväkseen, kun se pyrkii määrittämään, kuinka kieliopista voidaan johtaa annettu syöte. Näin jäsentäjästä saadaan tieto annetun syötteen oikeellisuudesta. [1]

Listaus 4.6: Makrokielen merkistö

```
CHARACTERS
letter = 'a'..'z' + 'A'..'Z' + '_'.
digit = '0'..'9'.
hexDigit = digit + "ABCDEF".
```

Listaus 4.7: Makrokielen päätesymbolit

```
identifier = letter {letter | digit}.
decNumber = digit {digit}.
hexNumber = "0x" hexDigit {hexDigit}.
```

Tätä rakennetta voidaan käyttää hyväksi myös makrokielen kanssa. Kielessä on hyvin määritelty erilaiset makrokomentojen konstruktiot, joista on helppo muodostaa kontekstiton kielioppi. Ajatuksena on luoda selaaja ja jäsentäjä makrokielelle *Coco/R* kääntäjägeneraattorilla [7]. Niiden avulla voidaan tarkistaa ohjelmistossa käytettyjen tai käyttäjän syöttämien makrokomentojen syntaksi.

Coco/R muodostaa selaajan DFA-automaattina (Determinstic finite Automaton), jolle määritellään päätesymbolit säännöllisellä EBNF-kieliopilla (Extended Backus-Naur-Form). Jäsentäjä määritellään joukolla EBNF-produktioita, joissa voidaan käyttää attribuutteja sekä syntaksiohjattua kääntämistä, mikä tarkoittaa, että kieliopin määrittelyyn voidaan sisällyttää Java-koodia, jolla voidaan tehdä esimerkiksi erilaisia semanttisia tarkistuksia. Kieliopin tulisi olla LL(1)-kielioppi, eli jäsentäjä lukee syötettä vasemmalta oikealle ja muodostaa vasemman puoleisen johdon, käyttäen yhden merkin eteenpäin kurkistusta (lookahead). [7]

Kielioppi määritellään MacroLanguageCompiler.ATG tiedostossa (attributed grammar). Ennen selaajan ja jäsentäjän määrittelyjä tiedostoon voidaan tarvittaessa liittää Java-koodia, joka käännetään jäsentäjän lopulliseen luokkaan. Sen jälkeen tiedostossa määritellään käytettävä merkistö ja päätesymbolit listausten 4.6 ja 4.7 mukaisesti. PRODUCTIONS osio sisältää produktiot konfigurointi- ja datansiirtokomennoille. Listaus 4.8 muodostaa produktion konfigurointikomennolle sekä produktiolle, jolla kuvataan makrokomennon suorituskertojen määrä. Listauksesta nähdään, kuinka syntaksiohjattua kääntämistä käytetään hyväksi produktioissa kirjoittamalla Javakoodi *(. .)*-rakenteen sisään.

Listaus 4.8: Makrokielen produktioita

```
ComfCmdDecl (. int num = 0; .)
=
"*"
CmdDecNumDecl <out num>
(. if (num == -1)
    SemErr("Invalid command decoder command.");
.)
"#"
.

ExecLoopCountDecl
=
","
(
identifier
(. if (t.val.startsWith("c")) {
        String count = t.val.substring(1);
        try {
            // Covert to integer and store that number
            int loops = Integer.parseInt(count);
            System.out.println("Loop count: " + loops);
        } catch (NumberFormatException error) {
            SemErr(count + " is not a valid loop count value.");
        }
    }
    else if (t.val.toLowerCase().startsWith("i")) {
        SemErr("Infinite execution cannot have loop count value");
    }
    else {
        SemErr(t.val + " is not a valid loop count value.");
    }
.)
|
// Infinite execution
"I"
|
// Time to run macro in seconds
decNumber
)
.
```

ATG-tiedosto kieliopille käännetään komennolla:

```
java -jar Coco.jar MacroLanguageCompiler.ATG,
```

joka luo *Scanner.java* ja *Parser.java* tiedostot. Näistä luokista voidaan muodostaa oliot ohjelmistoon makrokielen syntaksin tarkistusta varten. Coco/R käyttää *Parser.frame* ja *Scanner.frame* tiedostoja Java-luokkien muodostamiseen. Näissä tiedostoissa määritellään, että oletuksena kääntäjä lukee käännettävän datan tiedostosta ja että virheviestit tulostetaan *System.out*-tietovirtaan. Tässä ohjelmistossa makrokomennot muodostetaan ja palautetut virheviestit halutaan merkkijonoina. Muodostamalla uusi tietovirta *CompilerOutputStream*, joka laajentaa *OutputStream*-luokkaa, voidaan kääntäjän viestit koota merkkijonoksi *StringBuilder*-oliolla. Uusi tietovirta sisältää metodin, jolla voidaan kysyä virheviestit jäsentäjältä, jos käännös sisälsi virheitä. Vastaavasti, koska luotu selaaja lukee normaalisti datan tiedostosta, muodostetaan tietovirta *CompilerInputStream*, johon voidaan kirjoittaa käyttäjän antama syöte tai ohjelman generoima makrokomento merkkijonona. Selaaja lukee datan tästä tietovirrasta ja käsittelee sen, jonka jälkeen välittää syötteen jäsentäjälle.

Käyttöliittymään toteutettiin *MacroCommandViewController*-luokka, joka toteuttaa näkymän, jossa käyttäjä voi kirjoittaa makrokomentoja tai ladata tiedoston, joka sisältää nämä komennot. Mikäli makrokomentojen jäsennys sisälsi virheitä, ne näytetään käyttäjälle, muutoin komento lähetetään rajapintakortille. Vastaus rajapintakortilta näytetään myös näkymässä. Käyttäjällä on myös mahdollisuus lähettää makrokomento suoraan jäsentämättä sitä. Tästä on hyötyä erityisesti silloin, jos rajapintakortille ohjelmoitu makrokieli sisältää sellaisia makrokomentoja, joita ohjelmisto ei vielä tue.

4.4 USB-kommunikointi

Rajapintakortissa olevan USB-piirin kanssa voidaan kommunikoida ohjelmallisesti FTDI:n tarjoamien laiteajurien avulla. FTDI tarjoaa suljetun D2XX-rajapinnan [5], jonka avulla USB-laiteajurien kanssa voidaan kommunikoida Windows käyttöjärjestelmässä DLL-kirjaston (Dynamic Linked Library) tai Linux käyttöjärjestelmässä so-kirjaston (shared object) avulla. Koska Java on korkean tason ohjelmointikieli, siinä ei ole mahdollisuutta käsitellä USB-laitteen laiteajurien tarjoamia kirjastoja suoraan Javan luokkakirjaston kautta. Jotta FTDI:n laiteajurin kanssa voidaan kommunikoida, tarvitaan erillinen kääreluokka, jonka kautta voidaan kutsua Javan virtuaalikoneessa ajettavasta ohjelmakoodista virtuaalikoneen ulkopuolisia eli natiiveja kirjastoja. Tätä tarkoitusta varten on kehitetty JNI (Java Native Interface) [17]. JNI on kaksisuuntainen rajapinta, joka mahdollistaa natiivikoodin kutsumisen Java-sovelluksesta sekä Java-koodin kutsumisen natiivikoodista.

4.4.1 JNI

Tässä diplomityössä toteutettiin kommunikointiin vaadittava natiivikirjasto C-kielellä, jonka funktioiden avulla voidaan kutsua FTDI:n kirjaston funktioita ja käsitellä funktioiden parametreja sekä prosessoida paluuarvoja. JNI-rajapintaa käytetään määrittelemällä Java-kääreluokan metodit *native*-avainsanalla. Kun tämä Java-luokka käännetään *javac*-kääntäjällä ja käännetty tavukoodi annetaan parametrina *javah*-työkalulle, se luo .h otsikkotiedoston. Tiedosto sisältää funktioiden määrittelyt kaikille natiiveiksi määritellyille kääreluokan metodeille sekä luokan vakioille. Tämän jälkeen luodaan .c tiedosto, jossa toteutetaan otsikkotiedoston määrittelemät funktiot. Jokainen funktio saa ensimmäisenä parametrina *JNIEnv**-osoittimen, joka osoittaa JNI-funktiotaulukon osoittimeen. Jokainen funktiotaulukon alkio on osoitin JNI-funktioihin, joita natiivikoodin tulee käyttää käsiteltäessä virtuaalikoneen tietorakenteita, esimerkiksi käsiteltäessä parametrina annettuja olioita. Toinen parametri on *jobject*, jonka sisältö riippuu siitä, että onko natiivi metodi staattinen vai ei. Jos metodi on staattinen, parametri on referenssi luokkaan, jossa metodi on määritelty. Jos metodi on instanssimetodi (instance method), parametri on referenssi sitä kutsuneeseen olioon. Funktion loput parametrit ja paluuarvo vastaavat Java-kääreluokan natiivi metodin määrittelyä.

Koska Java-ohjelmointikielessä ei ole osoittimia, ohjelmoija ei voi viitata ohjelmakoodissa väärään muistialueeseen. Ohjelmalle varattu muisti ei yleensä jää vapauttamatta automaattisen roskienkerääjän ansiosta. Tämän tyylisiä ohjelmointivirheitä voi kuitenkin helposti esiintyä, kun käytetään JNI:tä [19]. Siksi USB-kirjaston suunnitteluun ja testaukseen pitää kiinnittää erityistä huomiota, jotta muistivuotoja ei syntyisi. Jos metodin parametreina on primitiivi tietotyyppejä, niiden käyttäminen on melko suoraviivaista, Java-kielen *int* tietotyyppiä vastaa *jint* ja *float* tietotyyppiä *jfloat*. Vastaavat määrittelyt löytyvät myös kuudelle muulle primitiivi tietotyypeille [9]. Parametreina annettavien olioiden käsittely poikkeaa kuitenkin primitiivi tietotyypeistä. Parametrina annettava olio on referenssi (C-kielen osoitin), joka viittaa JVM:n sisäiseen tietorakenteeseen. Näitä olioita pitää aina käsitellä edellä mainittujen JNI:n funktioiden avulla, jotka ovat saatavilla *JNIEnv**-osoittimen avulla (funktion ensimmäinen parametri). Listauksessa 4.9 on esitetty, miten *openEx()*-funktio käyttää referenssiä *jstring* tyyppiseen olioon. Referenssiä käytetään *GetStringUTFChars()*-funktion kanssa natiivin merkkijonon muodostamiseksi. Jos muisti ei riittänyt operaation, funktio palauttaa *NULL*-arvon. Useat JNI:n funktiot toimivat vastaavasti virhetilanteissa ja näiden virhetilanteiden käsittely jää aina ohjelmoijan vastuulle. Ohjelmoijan tulee myös kutsua *ReleaseStringUTFChars()*-funktiota, kun natiivia merkkijonoa ei enää tarvita. Jos näin ei tehdä, syntyy muistivuoto, koska merkkijonolle varattu muisti jää varatuksi. Eli varattu muisti pitää aina muistaa vapauttaa.

Listaus 4.9: JNI-esimerkki

```
JNIEXPORT void JNICALL Java_javad2xx_JavaD2xx_openEx(JNIEnv *env, jobject obj, jstring serial) {
...
const jbyte *str;
str = (*env)->GetStringUTFChars(env, serial, NULL);
if (str == NULL) {
  ProcessErrors(env, "open(): Error, failed to allocate memory for native
         serial number\n", NULL);
  return;
}

FT_STATUS status = FT_OpenEx((PVOID)str, FT_OPEN_BY_SERIAL_NUMBER, (FT_HANDLE)&handle);

(*env)->ReleaseStringUTFChars(env, serial, str);
...
```

Natiivikoodissa täytyy myös kiinnittää huomiota poikkeusten käsittelyyn. Java-koodissa tapahtuva poikkeus siirtää suorituksen automaattisesti poikkeuskäsittelijään eli try / catch-lohkoon. Natiivikoodissa tätä automatiikkaa ei ole. Kun poikkeus on heitetty *ThrowNew()*-funktiokutsulla, natiivikoodin täytyy palauttaa suoritus JVM:lle manuaalisesti [9]. Poikkeuksen heittämisen jälkeen JNI-funktioita ei saa kutsua, ennen kuin heitetty poikkeus on käsitelty ja nollattu.

4.4.2 JavaD2xx

Seuraavaksi kuvataan toteutettu *JavaD2xx*-kääreluokka sekä natiivikirjasto, joilla voidaan kommunikoida FTDI:n kirjastojen kanssa. Natiivikirjastosta esitetään muutaman eri funktion toteutus. *JavaD2xx.java* toteuttaa kääreluokan, joka määrittelee FTDI:n rajapinnan metodit. Metodeja on mm. kytkettyjen FTDI:n laitteiden listaamiseksi, yhteyden avaamiseksi ja sulkemiseksi sekä laitteelta lukemiseksi ja kirjoittamiseksi. Luokassa on myös kaksi staattista sisäluokkaa. Listauksessa 4.10 on esitelty D2XX-rajapinnan määrittelemä tietue *FT_DEVICE_LIST_INFO_NODE*, joka sisältää yhteen tiettyyn laitteeseen liittyviä tietoja, kuten esimerkiksi ID:n, laitteen kuvauksen ja sarjanumeron [5].

Listauksessa 4.11 olevaa staattista sisäluokkaa *DeviceInfoListNode* käytetään esimerkiksi silloin, kun kysytään listaus kytketyistä FTDI:n laitteista. Poik-

Listaus 4.10: *FT_DEVICE_LIST_INFO_NODE*-tietue

```
typedef struct _ft_device_list_info_node {
  DWORD Flags;
  DWORD Type;
  DWORD ID;
  DWORD LocId;
  char SerialNumber[16];
  char Description[64];
  FT_HANDLE ftHandle;
} FT_DEVICE_LIST_INFO_NODE;
```

keuksena FTDI:n tietueeseen on, että luokka sisältää myös laitteen kahvan (handle), joka saadaan laitteeseen yhteyttä avatessa. Luokan tietoja käytetään laitteen tunnistamiseen ohjelman alkuvalikossa. Toinen staattinen sisäluokka on *JavaD2xxException*, jonka toteutus on kuvattu listauksessa 4.12. Natiivikoodi heittää (throws) *JavaD2xxException*-luokkan poikkeuksen virhetilanteissa.

Listaus 4.11: *DeviceInfoListNode*-luokka

```
public static class DeviceInfoListNode {
  public int devFlags;
  public int devType;
  public int devId;
  public int devLocation;
  public String devSerialNumber;
  public String devDescription;
  public int devHandle;
}
```

Listaus 4.12: *JavaD2xxException*-luokka

```
public static class JavaD2xxException extends Exception {
  private static final long serialVersionUID = 1L;
  public JavaD2xxException(String msg) {
    super(msg);
  }
}
```

Listaus 4.13 sisältää esimerkkinä muutamien *JavaD2xx*-luokan metodien määrittelyjä. Näitä metodeja voidaan kutsua Java-koodista laitteen kanssa kommunikoimiseksi. Kaikki metodit heittävät JavaD2xxException poikkeuksen virhetilanteissa. Listaamalla FTDI:n laitteet *getDeviceInfoList()*-metodilla, saadaan lista kytketyistä laitteista, jonka indekseillä voidaan viitata tiettyyn laitteeseen. Yhteys voidaan avata antamalla tämä indeksi *open()*-metodille. *javah*-työkalu muodostaa *open()*-metodille listauksessa 4.14 olevan funktion prototyypin, joka sisältää paluuarvon tyypin (void), JNI-parametrit (JNIEnv* ja jobject) ja Java-luokan metodin parametrit (laitteen indeksin, (int)). Funktion nimi alkaa Java_ etuliitteellä, jota seuraa luokan ja metodin nimet alaviivalla erotettuina.

Listaus 4.13: *JavaD2xx*-luokan metodeja.

```
public static native DeviceInfoListNode[] getDeviceInfoList() throws JavaD2xxException;
private native void open(int devIndex) throws JavaD2xxException;
private native void openEx(String devSerialNum) throws JavaD2xxException;
public native void close() throws JavaD2xxException;
public native int read(byte[] buffer, int bytes) throws JavaD2xxException;
public native int writeBytes(byte[] b) throws JavaD2xxException;
```

Listaus 4.14: *open()*-funktion prototyyppi

```
JNIEXPORT void JNICALL Java_javad2xx_JavaD2xx_open(JNIEnv *, jobject, jint);
```

Funktion yksinkertaistettu toteutus on esitetty listauksessa 4.15. Ensin haetaan *JavaD2xx*-luokan arkkitehtuuriin sidonnainen *ftHandle* attribuutin arvo käyttäen *ftHandleID* referenssiä parametrina JNI:n funktiolle. Referenssi on alustettu funktiossa *JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *jvm, void *reserved)*, jota JVM kutsuu automaattisesti, kun natiivikirjasto ladataan. Tämä attribuuttiarvo on kahva, joka saadaan *FT_Open()*-funktiolta. Tätä kahvaa on käytettävä onnistuneen *FT_Open()*-funktion kutsun jälkeen kutsuttaessa muita FTDI:n funktioita, kunnes kyseinen laite on suljettu.

Kun laite on avattu, sille voidaan kirjoittaa ja siltä voidaan lukea dataa. Ohjelmistossa dataa lukeva säie kutsuu *read()*-metodia, jolle se antaa parametrina tavupuskurin sekä luettavien tavujen lukumäärän. Listauksessa 4.16 on pelkistetty *read()*-funktion toteutus. Ensimmäiseksi funktio hakee Java-luokalta kahvan. Sen jälkeen varataan muistia *malloc()*-funktiolla lukemiseen tarvittava määrä tavuja. *FT_GetQueueStatus()*-funktiolla kysytään laiteajurin vastaanottopuskurissa olevien tavujen lukumäärä. *GetByteArrayRegion()*-funktio kopioi parametrina annetun taulukon alueen edellä varattuun muistialueeseen. *FT_Read()*-funktio lukee x tavua vastaanottopuskurista kyseiseen muistialueeseen. *SetByteArrayRegion()*-funktio taas kopioi natiivista muistialueesta *FT_Read()*-funktion palauttaman lukumäärän tavuja säikeen antamaan tavupuskuriin (jbyteArray). Lopuksi natiivikoodissa varattu muisti vapautetaan. Koska dataa lukeva säie käyttää tavupuskuria tilapäisenä varastona merkkijonojen muodostamiseen, sitä ei tarvitse alustaa

Listaus 4.15: *open()*-funktion toteutus

```
JNIEXPORT void JNICALL Java_javad2xx_JavaD2xx_open(JNIEnv *env,jobject obj, jint devNum) {
#ifdef i386
  jint handle = (*env)->GetIntField(env, obj, ftHandleID);
#else
  jlong handle = (*env)->GetLongField(env, obj, ftHandleID);
#endif
  if (handle == INVALID_HANDLE) {

    FT_STATUS status = FT_Open(devNum, (FT_HANDLE)&handle);
    if (status == FT_OK) {
#ifdef i386
      (*env)->SetIntField(env, obj, ftHandleID, handle);
#else
      (*env)->SetLongField(env, obj, ftHandleID, handle);
#endif
    }
    else {
      ProcessErrors(env, "open(): Failed to open the device. Error %s\n",
        getErrorMessages(status));
    }
  }
  else {
    ProcessErrors(env, "open(): Error, device is not closed\n", NULL);
  }
}
```

erikseen. Tämän vuoksi *GetByteArrayRegion()*-funktiokutsu on jätetty kommentoiduksi, jotta taulukon kopiointia ei tehtäisi turhaan jokaisella funktion kutsulla.

JNI:tä käytettäessä kannattaa huomioida muutamia suorituskykyyn ja ohjelmiston vakauteen vaikuttavia asioita. Seuraavassa on listattu muutamia huomioita JNI:n käytöstä, joita Sheng Liang listaa kirjassaan [9]. Kuten edellä jo todettiin, ohjelmoijan tulee ennalta nähdä mahdolliset virhetilanteet ja ohjelmoida JNI-funktiot siten, että virhetilanteet tulevat käsiteltyä. Rajapintojen Java- ja natiivikoodin välillä tulisi olla mahdollisimman yksinkertaisia. Jos kontrollivuo kulkee jatkuvasti edes takaisin Java- ja natiivikoodin välillä, tämä voi vaikeuttaa ohjelmien ymmärtämistä ja ylläpitoa sekä estää joitakin tiettyjä kääntäjän tekemiä optimointeja. Eräs näistä on *inline*-optimointi, jossa funktiokutsu korvataan funktion toteutuksella. Natiivikoodin määrä tulisi myös minimoida, koska se voi aiheuttaa vaikeasti löydettäviä ohjelmointivirheitä. JNI-funktioiden parametrien oikeellisuus voidaan testata ajonaikaisesti antamalla JVM:lle käynnistyksessä argumentti *-Xcheck:jni*. Tällä argumentilla JVM tarkistaa JNI-funktioiden parametrit ja ilmoittaa mahdollisista virheistä. Tästä aiheutuu kuitenkin merkittävää kuormitusta (overhead), joten ominaisuus ei ole oletuksena käytössä. Natiivikoodi saa käyttöönsä luokan kentän (field) tai metodin ID:n kysymällä sitä JVM:ltä

Listaus 4.16: *read()*-funktion toteutus

```
JNIEXPORT jint JNICALL Java_javad2xx_JavaD2xx_read(JNIEnv *env, jobject obj,
                                                       jbyteArray dataStorage, jint bytes) {
  DWORD numBytesRead = -1;
#ifdef i386
  jint handle = (*env)->GetIntField(env, obj, ftHandleID);
#else
  jlong handle = (*env)->GetLongField(env, obj, ftHandleID);
#endif

  DWORD RxBytes = 0;
  FT_STATUS status;
  jbyte* nativeBuffer = (jbyte*)malloc(bytes * sizeof(jbyte));

  status = FT_GetQueueStatus((FT_HANDLE)handle, &RxBytes);
  if (status != FT_OK) {
    free(nativeBuffer);
    ProcessErrors(env, "read(): Failed to query queue status, error %s\n",
        getErrorMessages(status));
    return -1;
  }
  else {
    //(*env)->GetByteArrayRegion(env, dataStorage, 0, bytes, nativeBuffer);
    status = FT_Read((FT_HANDLE)handle, nativeBuffer, (DWORD)bytes, &numBytesRead);

    if (status != FT_OK) {
      free(nativeBuffer);
      ProcessErrors(env, "read(): Failed to read data from the device, error %s\n",
        getErrorMessages(status));
      return -1;
    }

    (*env)->SetByteArrayRegion(env, dataStorage, 0, bytes, nativeBuffer);
  }

  // Free preallocated memory
  free(nativeBuffer);

  return numBytesRead;
}
```

nimen ja tyypin kuvauksen perusteella. Natiivikoodin kannattaa puskuroida (cache) tarvittavat luokka, kenttä ja metodi ID:t heikkoihin globaaleihin referensseihin, jotta niitä ei tarvitse erikseen selvittää niitä käytettäessä. Listaus 4.17 kuvaa, miten *JavaD2xx*-kirjasto muodostaa globaalit referenssit tietyille ID:ille JNI_OnLoad()-funktiossa. Referenssin tulee olla heikko, jotta se voidaan poistaa muistista (unload). JNIEnv*-osoitinta ei tule puskuroida, koska se liittyy aina siihen säikeeseen, josta natiivikoodia kutsutaan. Näillä puskuroinneilla on huomattava vaikutus kirjaston suorituskykyyn, koska esimerkiksi *ftHandleID* ID:tä käytetään kahvan hakemiseksi kaikissa funktioissa, jotka käsittelevät tiettyä laitetta.

D2XX-rajapintaa käytettäessä kannattaa myös huomioida tiettyjen USB-

Listaus 4.17: ID:iden puskurointi

```
// Get class ID for JavaD2xx class
jclass cls = (*env)->FindClass(env, "Ljavad2xx/JavaD2xx;");
if (cls == NULL) {
        return JNI_ERR;
}

// Create weak global reference to be able to unload class
JavaD2xx = (*env)->NewWeakGlobalRef(env, cls);
(*env)->DeleteLocalRef(env, cls);
if (JavaD2xx == NULL) {
        return JNI_ERR;
}

// Create cached reference for ftHandle instance field
ftHandleID = (*env)->GetFieldID(env, JavaD2xx, "ftHandle", "I");
if (ftHandleID == NULL) {
        return JNI_ERR;
}

// Create cached reference for JavaD2xxException
cls = (*env)->FindClass(env, "Ljavad2xx/JavaD2xx$JavaD2xxException;");
if (cls == NULL) {
        return JNI_ERR;
}
JavaD2xxException = (*env)->NewWeakGlobalRef(env, cls);
(*env)->DeleteLocalRef(env, cls);
if (JavaD2xxException == NULL) {
        return JNI_ERR;
}

// Create cached reference for DeviceInfoListNode
cls = (*env)->FindClass(env, "Ljavad2xx/JavaD2xx$DeviceInfoListNode;");
if (cls == NULL) {
        return JNI_ERR;
}
DeviceInfoListNode = (*env)->NewWeakGlobalRef(env, cls);
(*env)->DeleteLocalRef(env, cls);
if (DeviceInfoListNode == NULL) {
        return JNI_ERR;
}
```

parametrien vaikutus, joka on kuvattu FTDI:n AN232B-04 sovellusohjeessa (application note) [4]. Nämä parametrit ovat USB-paketin koko ja USB-latenssiaika. USB-paketin koko on oletuksena 4096 tavua ja latenssiaika 16 millisekuntia. Nämä arvot ovat liian suuria lähetettävän datan kokoon nähden ja ne aiheuttavat ongelmia, joihin viitattiin jo luvussa 2.5. USB-tiedonsiirto ei käytä keskeytyksiä, kuten esimerkiksi sarjaportit, vaan se on toteutettu skedulerilla (scheduler), joka vastaanottaa dataa paketteina. Kun rajapintakortilta luetaan dataa, FTDI:n laiteajuri lähettää lukupyynnön (read request) USB-väyläohjaimelle (USB host controller). Väyläohjain lukee dataa 64 tavun paketeissa. Väyläohjain palauttaa luetut tavut FTDI:n laiteajurille vasta, kun jokin seuraavista ehdoista täyttyy:

- USB-paketti täyttyy
- Se vastaanottaa paketin, jonka pituus on < 64 tavua

USB-latenssiaikaa voidaan säätää funktiolla *FT_SetLatencyTimer()*. Jos väyläohjain vastaanottaa dataa hitaasti, eikä USB-paketti ole täynnä, voi latenssiajastin katkaista lukupyynnön tietyn ajan kuluttua lähettämällä paketin, jonka koko on pienempi kuin 64 tavua. Tällä parametrilla voidaan säätää datansiirron tasaisuutta yhdessä USB-paketin koon kanssa. USB-paketin kokoa voidaan pienentää *FT_SetUSBParameters()*-funktiolla. Keskimääräinen vastaanotettava tavujen määrä kommunikointikehykselle on noin 70 tavua. Pahimmassa tapauksessa FTDI:n laiteajuri palauttaa siis kerrallaan n. 56 kommunikointikehyksen verran dataa. Tämä aiheuttaa viiveitä *OutputDataEvent*-tapahtumiin, jonka voi nähdä siitä, että ohjelma jää odottamaan dataa pitkäksi aikaa. Pienentämällä USB-paketin kokoa ja latenssiaikaa sopivassa suhteessa saadaan dataa tasaisemmin ohjelmiston prosessoitavaksi.

Listaus 4.18: Natiivikirjaston lataus

```
rem Load correct version of dynamic link library in Windows
java -jar ./lib/TestBitness.jar

if %ERRORLEVEL% == 32 (
rem echo Loading 32-bit libraries...
set LIB_PATH=%DEMO_HOME%\lib\JavaD2xx\Windows\x86
)

if %ERRORLEVEL% == 64 (
rem echo Loading 64-bit libraries...
set LIB_PATH=%DEMO_HOME%\lib\JavaD2xx\Windows\x64
)

...

# Load correct version of shared library in Linux
java -jar ./lib/TestBitness.jar

bit=$?
if test $bit -eq 32
then
  export LIB_PATH="./lib/JavaD2xx/Linux/i386"
fi

if test $bit -eq 64
then
  export LIB_PATH="./lib/JavaD2xx/Linux/x86_64"
fi

...
```

Toteutetut natiivikoodit käännetään gcc-kääntäjällä DLL- tai so-kirjastoiksi *make*-tiedostojen avulla. Natiivikirjasto joudutaan kääntämään erikseen eri prosessoriarkkitehtuureille sekä eri käyttöjärjestelmille. *Makefile.conf*-tiedosto määrittelee käännöksen kohteen sekä JDK:n (Java Development Kit), gcc:n ja FTDI:n ajureiden käännöksessä tarvittavat polut. *Makefile*-tiedosto sisältää ohjeet käännöksen tekemiseen. Käännöksen tuloksena saadaan JAR-tiedosto ja dll- tai so-tiedosto käyttöjärjestelmästä riippuen. Koska kirjastosta käännetään kohdejärjestelmän arkkitehtuuriin sidonnaisia versioita, tämä rikkoo ohjelman siirrettävyyden. Ongelma voidaan kuitenkin kiertää määrittämällä virtuaalikoneen arkkitehtuurin mukaan oikea polku natiivikirjastoille *-Djava.library.path*-argumentilla ohjelman käynnistyessä. Tässä ongelmana on se, että käynnistyksen hoitavassa skriptitiedostossa joudutaan selvittämään Java:n virtuaalikoneen data malli (data model) eli sen, onko virtuaalikone 32- vai 64-bittinen, jotta oikea natiivikirjasto saadaan ladattua. Listauksesta 4.18 voidaan nähdä, kuinka tämä tehdään eri käyttöjärjestelmissä. Kun ohjelmistossa viitataan ensimmäinen kerran *JavaD2xx*-luokkaan, luokan staattinen alustuslohko kutsuu *System.loadLibrary(String)*-metodia ladatakseen kirjaston. Tämän jälkeen virtuaalikone etsii natiivikirjastoa määritellystä polusta.

Luku 5

Arviointi

Diplomityön tavoitteena oli suunnitella ja toteuttaa ohjelmisto, johon voidaan mm. helposti lisätä uusia antureita ja anturityyppejä. Tässä luvussa arvioidaan suunnitelmaa ja toteutunutta järjestelmää luvussa 3.1 asetettuja vaatimuksia vasten.

Asetettuihin tavoitteisiin kuului myös ohjelmiston helppo asennus ja sen siirrettävyys eri käyttöjärjestelmille. Ohjelmisto paketoidaan ZIP-pakettiin, joka on käyttäjän helposti purettavissa, koska jokin ZIP-tiedostoformaattia tukeva ohjelma on valmiiksi asennettuna kaikissa moderneissa käyttöjärjestelmissä. Tämä operaatio ei myöskään vaadi pääkäyttäjän oikeuksia, jolloin ohjelma voidaan asentaa tavallisena käyttäjänä olettaen, että FTDI:n laiteajurit ovat asennettuina (kts. 5.4). Laiteajurien asennus vaatii pääkäyttäjän oikeudet. Pääkäyttäjän oikeudet tarvitaan myös ohjelman käynnistämiseen Linux-käyttöjärjestelmässä, jos sarjaporttimoduulit ovat ladattuina. Ohjelmisto toimii kaikilla olemassa olevilla rajapintakortin sukupolvilla, kuten vaadittiin. Tästä on tosin haittaakin, koska rajapintakorttien tukemat makrokielen versiot poikkeavat toisistaan, mikä monimutkaistaa järjestelmää. Muutoksia tehtäessä täytyy olla tarkkana, ettei toiminnallisuus rikkoudu jonkin rajapintakortin kanssa esimerkiksi niin, että jäädään odottamaan vastausta, vaikka kyseinen versio ei sitä palautakkaan annettuun komentoon.

Ohjelmistoarkkitehtuurin suunnittelussa käytettiin apuna suunnittelumalleja. Näitä malleja käyttämällä saatiin eri järjestelmän komponenttien välisiä yhteyksiä minimoitua. Sovelluslogiikalla on mahdollisuus tehdä takaisinkutsuja käyttöliittymään *DemoGUI*-rajapinnan kautta, esimerkiksi ilmoittaakseen erilaisista virheilmoituksista. Tämä tekee logiikan riippuvaiseksi rajapinnasta. Ongelmaa olisi voitu kiertää sillä, että käyttöliittymä rekisteröityisi kuuntelemaan virheilmoituksia tai muita tarvittavia toimenpiteitä. Näin sovelluslogiikka voisi ilmoittaa rekisteröityneelle kuuntelijalle kaikista tapahtumista niiden esiintyessä eikä se olisi riippuvainen *DemoGUI*-rajapinnan toteuttavista olioista.

Antureiden ominaisuudet kuvattiin Java:n *Properties*-luokan avulla. Luokka tarjoaa yksinkertaisen ja tehokkaan tavan käyttää avain–arvo-pareja ohjelman tai anturin ominaisuuksien tallentamiseksi. Näin voitiin toteuttaa geneeriset komponentit datan käsittelemiseksi, koska komponentit voivat käyttää avainta anturikohtaisen arvon hakemiseksi. Anturin rekisterit kuvattiin helposti luettavalla XML-kielellä. Käyttöliittymään saatiin toteutettua hyödylliset näkymät, jotka ovat helposti generoitavissa ohjelmallisesti, ja joissa käyttäjä voi tarkastella anturin rekistereitä jopa ilman muita teknisiä kuvauksia. Tällä hetkellä ohjelmistossa ei kuitenkaan ole tapaa kuvata näiden XML-tiedostojen avulla esimerkiksi epäsuoraa rekisteriosoitusta tai tilannetta, jossa rekisterin sisältö on erilainen luku- ja kirjoitusoperaatiolle.

Makrokielen jäsentäjällä on helppo jäsentää annettuja makrokomentoja. Jäsentäjän generoimiseen käytettiin *Coco/R* kääntäjägeneraattoria, jolla voidaan kieliopin määrittelyllä generoida selaajan ja jäsentäjän ohjelmakoodi ilman ohjelmointia. Koska makrokielen syntaksi on melko yksinkertainen, selaaja ja jäsentäjä olisi voitu luoda myös käsin. Toisaalta nyt uusien komentojen lisääminen makrokieleen on todella yksinkertaista ja uudet luokat kääntäjälle saadaan generoitua helposti.

USB-kirjaston suunnittelussa otettiin huomioon tiettyjä menetelmiä suorituskyvyn parantamiseksi, joita Sheng Liang on listannut kirjassaan [9]. Näin saatiin toteutettua vakaa ja tehokas kirjasto, jolla voidaan tehdä kutsuja natiivikoodiin. Koska kirjaston natiivikoodi joudutaan kääntämään erikseen eri käyttöjärjestelmille, tästä aiheutuu ylimääräisiä ongelmia, joita jouduttiin huomioimaan ohjelman käynnistymisen yhteydessä. Erilaisilla skriptitiedostoilla selvitetään käytettävän virtuaalikoneen data malli, jotta ladataan oikeat arkkitehtuurikohtaiset natiivikirjastot. USB-kirjaston luotettavuuden varmistamiseksi, sitä testattiin TIMESTAMP-makron avulla. Tällä makrolla saadaan jokaiselle näytteelle aikaleima, joiden perusteella voidaan päätellä, onko kaikki näyteet vastaanotettu oikein. Näytteet tallennettiin tiedostoon, josta aikaleimat myöhemmin testattiin.

5.1 Laajennettavuus

Seuraavissa luvuissa kuvataan sekä arvioidaan järjestelmän laajentamista erilaisten toimintojen osalta.

5.1.1 Uuden anturityypin lisääminen

Olemassa olevaan anturiperheeseen voidaan lisätä uusi anturityyppi helposti tekemättä muutoksia ohjelmakoodiin. Anturityyppi ja tyypin ID lisätään *xxx_product_codes.properties*-tiedostoon, missä xxx on anturiperheen nimi. *etc/data/xxx*-kansioon lisätään anturityypille parametritiedosto *y.properties*, missä y on anturin tyyppi, esimerkiksi SCC1300-D04. Tiedoston tulee sisältää anturiperheeseen sidonnaiset anturityypin parametrit, kuten herkkyydet ja mittaussuunnat. Muita muutoksia uuden anturityypin lisäämiseen ei tarvita, jos se ei sisällä sellaisia uusia operaatioita, joille vaaditaan toteutus *xxxDemoSystem*-luokkaan. Tällaisia operaatioita voivat olla esimerkiksi tarve alustaa jokin rekisteri tietyllä tavalla. Kun anturiperhe valitaan käyttöliittymästä, uusi anturityyppi listataan automaattisesti muiden anturiperheen anturityyppien kanssa. Jos kyseinen anturityyppi valitaan, järjestelmä lataa anturin parametrit lisätystä parametritiedostosta.

5.1.2 Uuden anturiperheen lisääminen

Uuden anturiperheen lisäämiseen vaaditaan hieman enemmän toimenpiteitä kuin pelkän anturityypin lisäämiseen. Kun järjestelmään halutaan lisätä uusi anturi, ohjelmoijan täytyy luoda oikea hakemistorakenne anturille ja lisätä anturin tarvitsemat tiedostot lisättyihin hakemistoihin. Ohjelmoijan tulee myös toteuttaa *fi.murata.ds.sensors*-pakettiin luokka uudelle anturille, joka periytyy jostain abstraktista anturi-luokasta. Esimerkiksi kiihtyvyysanturin tulee laajentaa *AccelerometerSensor*-luokkaa. Luokan tehtävänä on tarjota makrokomennot, joita tulee käyttää anturin rekistereiden lukemiseen ja kirjoittamiseen sekä metodi, jolla voidaan jäsentää anturilta mitattua ulostulodataa. Seuraavaksi ohjelmoijan tulee luoda luokka xxxDemoSystem *fi.murata.ds.demos.system*-pakettiin, joka periytyy *AbstractDemoSystem*-luokasta. Luokan tehtävänä on hoitaa rajapintakortin alustus kyseistä anturia varten, alustaa tarvittavat anturin rekisterit sekä tarjota metodit kyseisen anturin kontrolloimiseksi, esimerkiksi rekistereiden lukemiseksi ja kirjoittamiseksi. Eli *fi.murata.ds.sensors*-paketin luokat tietävät komennot, joilla anturin kanssa kommunikoidaan ja *fi.murata.ds.demos.system*-paketin luokat tietävät tavan, jolla kommunikointi tulee tehdä. Esimerkkinä voidaan antaa sarjanumeron lukeminen SCC2XXX-antureille. *SCC2xxx*-luokka toteuttaa *getReadRegisterMacros()*-metodin, joka palauttaa makrokomennon parametrina annetun rekisterin lukemiseksi. *SCC2xxxDemoSystem*-luokka toteuttaa *readSerialNumber()*-metodin, joka tietää mistä rekistereistä sarjanumero kyseiselle anturille luetaan. Luokka osaa myös käyttää rajapintakortin toteuttavaa *DemoBoard*-luokkaa kyseisen komennon lähettämiseen ja vastauksen lukemiseen.

Jos kaikki anturin sisältämät ominaisuudet ovat tuettuja käyttöliittymässä, yllä kuvatut operaatiot riittävät uuden anturin lisäämiseen järjestelmään. Jos anturi kuitenkin sisältää sellaisia ominaisuuksia, joita käyttöliittymään ei vielä ole toteutettu, tämä saattaa monimutkaistaa anturin lisäämistä. Järjestelmään saatetaan tarvita kokonaan uusi näkymä tai vanhaa näkymää joudutaan muokkaamaan. Toisaalta käyttöliittymää ei voida suunnitella niin, että se sisältäisi valmiiksi kaikki mahdolliset ominaisuudet, vaan se kehittyy jokaisella iteraatiolla antureita lisättäessä. Tässä tapauksessa suurin osa toteutukseen kuluvasta ajasta menee yleensä uuteen anturiin tutustuessa.

5.1.3 Uuden rajapintakortin lisääminen

Uusi rajapintakortti voidaan lisätä periyttämällä uusi luokka abstraktista *DemoBoard*-luokasta *fi.murata.ds.devices.demoboard*-pakettiin. Luokan tulee määritellä rajapintakortin pinnit sekä toteuttaa tietyt metodit, kuten *initialize()* ja *getChipSelectNPin()*, jolla voidaan tehdä tarvittavat alustukset sekä kysyä pinnin numero tietylle CSB-signaalille. Luokka voi sisältää myös metodeja, jotka ovat olemassa vain kyseiselle rajapintakortille. Luokalle pitää myös lisätä määrittely *DemoBoardID*-literaaliluokkaan sekä muokata *getBoardIds()*-metodia, jolla *AbstractDemoSystem*-olio listaa käytetyt rajapintakortit. Jos rajapintakortin kuva halutaan näyttää alkuvalikossa, sille pitää lisätä kuva *etc/images* kansioon ja muokata *MFIDemo.loadBoardImage()*-metodia, jotta kuva ladataan oikein.

Yllä oleva kuvaus pätee vain, jos uusi rajapintakortti toimii myös makrokomennoilla, koska ohjelmistoon toteutettu rakenne olettaa rajapintakorttien käyttävän makrokomentoja. Tällä hetkellä tämä ei ole ongelma, koska kaikki rajapintakortin sukupolvet käyttävät makrokomentoja, mutta tulevaisuudessa tähän saatetaan joutua tekemään muutoksia.

5.1.4 Uuden näkymän lisääminen

Uudelle näkymälle lisätään määrittely *fi.murata.ds.demos.gui.view*-paketista löytyvään *ViewTypes*-literaaliluokkaan. *View*-luokka sisältää staattiset metodit näkymien listaamiseksi sekä otsikoiden määrittelemiseksi kullekin näkymälle erikseen. Tämän jälkeen luodaan näkymäohjain periyttämällä *ViewController*-luokasta uusi luokka. Jos näkymässä on tarkoitus näyttää kuvaajia, voidaan näkymäohjain periyttää *ChartViewController*-luokasta, joka sisältää tiettyjä kuvaajiin liittyviä attribuutteja ja metodeja. Jos näkymäohjain haluaa vastaanottaa ulostulodataa, sen tulee toteuttaa *OutputDataEventListener*-rajapinta. Muuten näkymäohjaimen toteutus riippuu sen käyttötarkoituksesta.

5.2 Ohjelmiston testaus

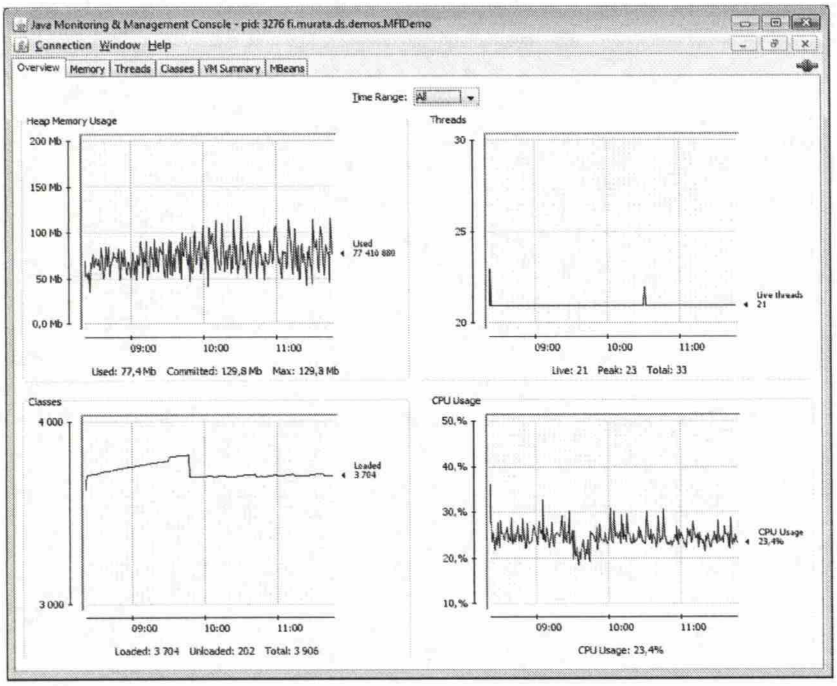
Ennen ohjelmiston vertailua vanhaan LabVIEW-ohjelmistoon, toteutettua MFIDemo-ohjelmistoa ja etenkin sen prosessorin ja muistin käyttöä testattiin *JConsole*-työkalulla, joka on osa JDK:ta. Ohjelmiston testaus on tärkeää sen luotettavan toiminnan varmistamiseksi.

Taulukko 5.1: Testausympäristö

Suoritin	Intel Core(TM) i5 M520 @ 2,4 GHz
Keskusmuisti	4 GB
Käyttöjärjestelmä	Windows 7 Enterprise (32-bit) SP1
Java JRE & JDK	Java 7 Update 17
FTDI:n ajuriversion	2.8.28.0
LabVIEW RTE	8.5

JConsole täyttää JMX-spesifikaation (Java Management Extensions), jonka avulla voidaan toteuttaa erilaisia hallinta- ja monitorointi-sovelluksia Java-sovelluksille. JConsole-työkalulla voidaan kerätä informaatiota Java-sovellusten suorituskyvystä sekä resurssien käytöstä, kun sovellusta ajetaan Java:n virtuaalikoneessa. Ohjelmiston testauksessa käytettiin HP:n Elitebook 8440p kannettavaa tietokonetta. Taulukossa 5.1 on testauksessa käytetty testiympäristö.

JConsole jaottelee ohjelmiston käyttämän muistin pinomuistiin (heap) ja muuhun muistialueeseen (non-heap) sekä erilaisiin erikseen nimettyihin kiinteisiin muistialueisiin (memory pool), joiden olemassaolo riippuu käytettävästä virtuaalikoneesta. Pinomuistiin tallennetaan virtuaalikoneen ajonaikana luomat luokkien instanssit ja taulukot. Automaattisen roskienkerääjän tehtävänä on yrittää vapauttaa muistia pinosta poistamalla sieltä olioita, joihin ei enää viitata ohjelmassa. JConsole jaottelee muistin erilaisiin sukupolviin (generation) riippuen siitä, miten kauan ne ovat olleet pinossa ja miten ne ovat selvinneet automaattisesta roskienkeruusta. Testatessa näistä sukupolvista tulisi seurata etenkin *"CMS Old Gen"*-sukupolvea, joka kuvaa olioita, jotka ovat toistuvasti selvinneet roskienkeruusta. Tämän muistialueen kasvaminen voi olla merkki muistivuodosta. Automaattinen roskienkerääjä vapauttaa yleensä olioille varatun muistin, mutta jos olioon jää viittauksia, vaikka sitä ei enää tarvita, muisti jää edelleen varatuksi. Tämä saattaa lopulta aiheuttaa pinon ylivuodon, jos pinon koko on kiinteästi määritelty tai käytettävä keskusmuisti loppuu kesken. [15]



Time Range: All

Used: 77,4 Mb Commited: 129,8 Mb Max: 129,8 Mb

Live: 21 Peak: 23 Total: 33

Loaded: 3 704 Unloaded: 202 Total: 3 906

CPU Usage: 23,4%

Kuva 5.1: JConsole-työkalun yleisnäkymä

JConsole-työkalun alkuvalikosta valitaan monitoroitavan prosessin nimi ja sen PID (Process IDentifier). Kun yhteys Java-sovellukseen on muodostettu, JConsole näyttää kuvassa 5.1 olevan näkymän, josta voi seurata ohjelmiston prosessorin sekä pinomuistin käyttöä ja ladattujen luokkien sekä luotujen säikeiden lukumääriä. *Heap Memory Usage*-kuvaajassa näkyy ohjelmiston käyttämä pinomuisti. Kuvaajasta nähdään selvästi, miten roskienkerääjä on poistanut olioita tietyn väliajoin ja miten pinomuistin koko on asettunut tietylle välille. *CPU Usage*-kuvaajasta nähdään prosessorin kuormitus. Matalat arvot kuvaajassa ovat ajalta, jolloin ohjelmisto ei piirtänyt kuvaajia ulostulodatalle ja korkeammat ajalta, kun kuvaajat olivat näkyvissä. Näistä nähdään, että prosessorin kuormitus pysyy melko tasaisena koko ohjelman suorituksen ajan. Noin 5% prosessorikuormasta kuluu kuvaajien piirtämiseen. Testin aikana ohjelmisto keräsi anturilta 2000 näytettä (samples) sekunnissa ja piirsi ne kuvaajaan, laski 100:n näytteen liukuvan keskiarvon sekä keskihajonnan

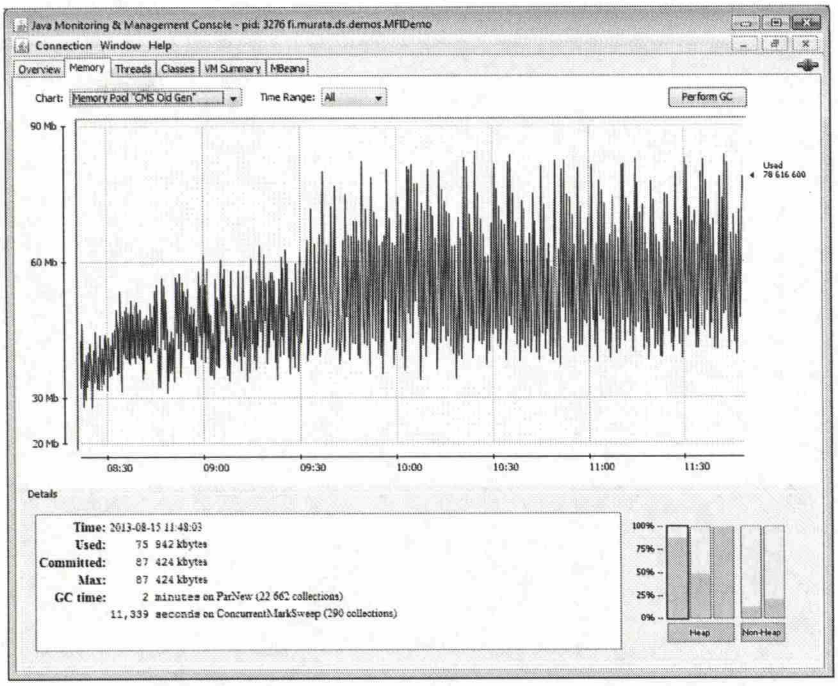


Chart: Memory Pool "CMS Old Gen"

Time Range: All

```
     Time: 2013-08-15 11:48:03
     Used:     75 942 kbytes
Committed:     87 424 kbytes
      Max:     87 424 kbytes
  GC time:      2 minutes on ParNew (22 662 collections)
           11,339 seconds on ConcurrentMarkSweep (290 collections)
```

Kuva 5.2: JConsole-työkalun "*CMS Old Gen*"-näkymä

5000:lle näytteelle. Kuvassa 5.2 esitetään, miten *"CMS Old Gen"*-muistialue käyttäytyy ajan kuluessa. Kuvaajasta voidaan päätellä, että ainakaan tässä testiympäristössä muistivuotoja ei esiintynyt.

5.3 Ohjelmistojen vertailua

LabVIEW-ohjelmisto SCC1300-anturille on nimeltään "Combined gyroscope and ACC" ja sen käyttöliittymä on esitetty kuvassa 2.6. Kuvassa 5.3 on vastaava näkymä MFIDemo-ohjelmistolle. LabVIEW-ohjelmisto käyttää makrokomentoja, joilla dataa saadaan anturilta noin 12 millisekunnin välein, eli noin 83 kertaa sekunnissa. MFIDemo-ohjelmistossa parametrit säädettiin vastaamaan mahdollisimman hyvin LabVIEW-ohjelmiston asetuksia, jotta tuloksia voidaan verrata keskenään. Koska JConsole-työkalu on Java-prosessien monitorointiin, ohjelmistojen resurssien käyttöä vertailtiin *Performance Monitor* nimisellä Windows-sovelluksella. Sovelluksessa käytettiin oletusasetuksia, jolloin se näyttää keskiarvon prosessorin kuormitukselle 1 minuutin ja 40 sekunnin ajalta.

Taulukko 5.2: Testitulokset

Testi	min	avg	max
MFIDemo (CPU)	7,8	17,8	26,7
LabVIEW (CPU)	14,0	23,7	29,6
MFIDemo (Muisti)	-	114,9	-
LabVIEW (Muisti)	-	27,0	-

Taulukosta 5.2 olevista tuloksista nähdään, että prosessorin kuormituksessa (CPU) ei ole merkittäviä eroja ohjelmistojen välillä, kun käytetään samoja asetuksia. Uusi MFIDemo käyttää kuitenkin aika paljon enemmän muistia kuin vanha ohjelmisto. Tämä johtuu siitä, että ohjelma varaa suuria muistialueita esimerkiksi keskiarvon ja keskihajonnan laskentaa varten jokaiselle akselille. Jos parametri keskihajonnan laskemiseksi on pois päältä, laskentaa ei tehdä, mutta muisti pysyy kuitenkin varattuna laskennalle.

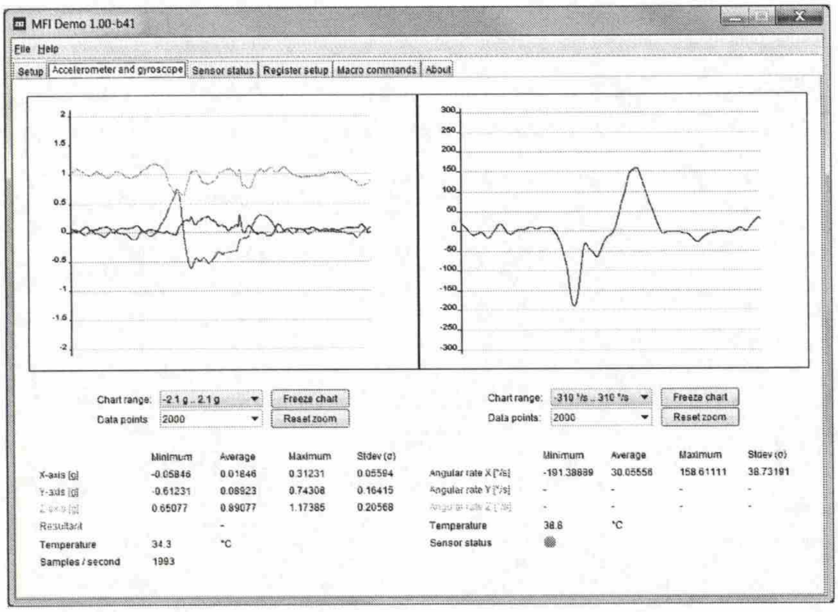


Chart range: -2.1 g .. 2.1 g
Data points: 2000

Chart range: -310 °/s .. 310 °/s
Data points: 2000

	Minimum	Average	Maximum	Stdev (σ)
X-axis [g]	-0.05846	0.01846	0.31231	0.05594
Y-axis [g]	-0.61231	0.08923	0.74308	0.16415
Z-axis [g]	0.65077	0.89077	1.17385	0.20568
Resultant		-		
Temperature	34.3	°C		
Samples / second	1993			

	Minimum	Average	Maximum	Stdev (σ)
Angular rate X [°/s]	-191.38889	30.05556	158.61111	38.73191
Angular rate Y [°/s]	-	-	-	-
Angular rate Z [°/s]	-	-	-	-
Temperature	38.8	°C		
Sensor status	●			

Kuva 5.3: MFIDemo-ohjelmiston näkymä ulostulodatalle

MFIDemo-ohjelmistossa on paljon enemmän ominaisuuksia kuin vanhassa ohjelmistossa. Näitä ominaisuuksia on vertailtu taulukossa 5.3. Jos kaikki mahdolliset ominaisuudet otetaan käyttöön ja kuvaajiin lisätään maksimimäärä näytteitä, prosessorin kuormitus kasvaa merkittävästi. Keskiarvo (avg) kasvoi tässä tapauksessa 17,8 prosentista 96,8 prosenttiin.

Taulukko 5.3: Ohjelmistojen ominaisuudet

Ominaisuus	MFIDemo	LabVIEW
Kuvaaja kiihtyvyysanturin akseleille	X	X
Kuvaaja kulmanopeusanturin akselille	X	X
Numeeriset lukuarvot	X	X
Säädettävä kuvaajan raja-arvot	X	X
Säädettävä keskiarvon laskenta	X	-
Keskihajonnan laskenta	X	-
Mahdollisuus suurentaa kuvaajaa (zoom)	X	-
Datan tallennus tiedostoon	X	-
Tilabittien valvonta	X	-
Virransäästötilojen asetus	X	-
Rekistereiden luku ja kirjoitus	X	-
Mahdollisuus lähettää makrokomentoja	X	-
Näyttää anturin lämpötilat	X	-
Näyttää anturin revision ja sarjanumeron	X	-

5.4 Kehitys tulevaisuudessa

Työssä toteutettiin Java-ohjelmisto, jota jaetaan asiakkaille ZIP-pakattuna tiedostona. Paketti sisältää ohjelmiston vaatiman hakemistorakenteen ja tiedostot. Tämä toimii muuten hyvin, mutta se mahdollistaa käyttäjän tekemät muutokset ohjelmistoon ja sen paketoimisen uudelleen, eikä se ei ota kantaa kohdejärjestelmän konfiguraatioon. USB-kirjastosta joudutaan kääntämään arkkitehtuurikohtaiset kirjastot, joiden lataaminen käynnistyksessä voi epäonnistua. Eräs vaihtoehtoinen jakelutapa olisi toimittaa Linux-jakelu asiakkaalle, johon demonstrointi järjestelmän ohjelmisto kirjastoineen olisi valmiiksi asennettuna. Näin tiedettäisiin kaikki ohjelmistokomponentit ja niiden vaikutus demonstrointi järjestelmän ohjelmistoon. Tässä ongelmana taas on se, että ei ole mitään kovin suoraviivaista mahdollisuutta siirtää Linux-jakelua USB-muistille, josta asiakas voisi jakelun käynnistää (Live-CD). Yksi ratkaisu olisi toimittaa valmiiksi konfiguroitu USB-muisti rajapintakortin mukana. Tämä ei kuitenkaan haluta tehdä, koska se nostaa järjestelmän kokonaishintaa eikä se mahdollista ohjelmiston päivitysten jakelua Murata:n www-sivujen kautta. Linux-jakelusta riippuen, Live-CD-tyyppinen ratkaisu saattaa myös kuormittaa liikaa vanhoja tietokoneita, joten tämän ratkaisun kanssa ohjelmiston suorituskyky saattaa kärsiä.

Toinen vaihtoehto perinteiselle ZIP-paketille on Java:n Web Start teknologia, joka mahdollistaa Java-sovellusten latauksen ja suorituksen suoraan www-sivuilta ilman erillistä asennusta. Asiakas voisi käynnistää ohjelmiston JNLP-protokollan (Java Network Launching Protocol) avulla avaamalla yrityksen verkkosivuilta linkin, joka sisältää listauksessa 5.1 näkyvän koodin. JNLP:tä käytettäessä ohjelmisto tulee paketoida kokonaisuutena JAR-tiedostoihin, jotka ladataan kokonaan asiakkaan koneelle silloin, kun linkki avataan. Samalla voidaan tarkistaa vaadittu JRE:n versio sekä ladata uusin versio ohjelmistosta, jos se on päivittynyt. Tämä lähestymistapa ei siis vaatisi erillistä ohjelmiston asennusta ja asiakas saisi aina uusimman version ohjelmistosta. JNLP:n avulla ladattu Java-sovellus toimii oletuksena erillisessä rajoitetussa ympäristössä (sandbox). Rajoitusten poistamiseksi ladattava JAR-tiedosto tulee digitaalisesti allekirjoittaa *jarsigner*-työkalulla, joka on osa JDK:ta. Muuten ohjelmistolla ei ole pääsyä esimerkiksi levyjärjestelmään eikä näin ollen tiedostojen tallennus onnistu. Tässä vaihtoehdossa täytyy myös ottaa huomioon kolmannen osapuolen avoimien kirjastojen lisenssiehdot. Asiakkaalla tulee olla mahdollisuus vaihtaa tai muokata käytettävää kirjastoa LGPL-lisenssin mukaisesti. [16]

Listaus 5.1: JNLP-esimerkki

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jnlp spec="1.0+" codebase="http://www.muratamems.fi/demos" href="">
    <information>
        <title>MFI Demo application</title>
        <vendor>Murata Electronics Oy</vendor>
        <offline-allowed/>
    </information>
    <resources>
        <j2se version="1.6+" href="http://java.sun.com/products/autodl/j2se"/>
        <jar href="MFIDemo.jar" main="true" />
    </resources>
    <applet-desc
         name="MFI Demo" main-class="fi.murata.ds.demos.MFIDemo"
         width="1000" height="700">
    </applet-desc>
  <update check="background"/>
</jnlp>
```

Työssä rajattiin pois ohjelman kahden eri prosessin välisen etärajapinnan toteutus. Tulevaisuudessa myös tämä rajapinta voidaan kehittää, jos se nähdään tarpeelliseksi. Myös edellä kuvattuja paketointiratkaisuja kannattaa edelleen miettiä ja valita sitten jokin ZIP-pakettia modernimpi vaihtoehto. Toimittamalla demonstrointi järjestelmän ohjelmisto osana Linux-jakelua poistaisi monta käyttöympäristöön liittyvää ongelmaa.

Luku 6

Yhteenveto

Murata Electronics Oy:n kehittämiä MEMS-teknologiaan perustuvia antureita käytetään monissa erilaisissa sovelluksissa ja uusia antureita kehitetään jatkuvasti. Näiden antureiden demonstroimiseksi tai evaluoimiseksi tarvitaan tämän mahdollistama demonstraatio järjestelmä ohjelmistoineen. Vanhojen ohjelmiston päivitykselle nähtiin tarve ja niinpä järjestelmälle päätettiin suunnitella kokonaan uusi ohjelmisto, jota voidaan käyttää kaikkien antureiden kanssa.

Tämän diplomityön tarkoituksena oli suunnitella asetetut vaatimukset täyttävä ohjelmistoarkkitehtuuri ja toteuttaa se Java-ohjelmointikielellä. Ohjelmistolle määriteltiin tietyt toiminnalliset vaatimukset. Näihin vaatimuksiin kuului esimerkiksi tietynlaisten näkymien toteutus. Näkymien haluttiin helpottavan käyttäjän tekemiä toimenpiteitä. Käyttäjän tulisi voida esimerkiksi tutkia anturin toimintaa ja tarkastella sen sisäisiä rekistereitä yksityiskohtaisien kuvauksien kanssa sekä kirjoittaa ja lähettää makrokomentoja rajapintakortille niin, että järjestelmä osaa tarkistaa nämä komennot.

Tärkeimmät ei-toiminnalliset vaatimukset liittyivät ohjelmiston helppoon käytettävyyteen sekä laajennettavuuteen, pääasiassa uusien anturien lisäämiseen ohjelmistoon. Koska eri anturiperheiden sisäinen toteutus poikkeaa paljon toisistaan, tämän työn suurimmat haasteet liittyivätkin juuri antureiden ominaisuuksien määrittelyyn niin, että ohjelmistossa voitiin käyttää yleiskäyttöisiä komponentteja kaikkien antureiden ohjaukseen ja datan esittämiseen.

Ohjelmiston suunnittelussa lähdettiin liikkeelle käyttäen perinteisiä olio-ohjelmoinnin menetelmiä. Ensin mietittiin ohjelmiston arkkitehtuuria, jossa käytettiin hyväksi erilaisia suunnittelumalleja. Tämän jälkeen hahmoteltiin suunniteltavaa järjestelmää osittamalla järjestelmää sanaston ja käyttötapauksien avulla. Tämän osituksen lopputuloksena saatiin hierarkinen ja helposti laajennettava luokkamalli, jota lähdettiin toteuttamaan. Toteutuksen aikana luokkamallia tarkennettiin iteraatioiden välillä. Ohjelmiston testauksella pyrittiin varmistamaan sen vakaa ja vaatimukset täyttävä toiminta. Testeillä ei löydetty muistivuotoja tai ongelmia, jotka aiheuttaisivat ylimääräistä prosessorin kuormitusta. Kommunikaation testaus ei osoittanut merkkejä tiedon katoamisesta.

Diplomityössä onnistuttiin toteuttamaan hyvin vaatimukset täyttävä ohjelmisto. Erilaiset hyödylliset näkymät datan visualisoimiseksi, antureiden määrittely järjestelmään XML-kielellä, makrokielen jäsentäjän toteuttaminen sekä USB-laitteiden kommunikointia varten suunniteltu kirjasto täyttivät suurimman osan asetetuista vaatimuksista. Ohjelmistosta saatiin myös paljon monipuolisempi ja hyödyllisempi verrattuna aiempiin LabVIEW-ohjelmistoihin. Toisaalta, monipuolisten ominaisuuksien vuoksi, se vaatii myös enemmän resursseja suorituksen aikana kuin edeltäjänsä. Suunnittelussa ja toteutuksessa onnistuttiin hyvin, mutta kuten edellisessä luvussa todettiin, ohjelmistossa on kuitenkin vielä muutamia kehityskohteita.

Kirjallisuutta

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools*, second ed. Pearson Education, Inc, 2007.

[2] Continental Automotive. Electronic stability control. `http://www.conti-online.com/generator/www/de/en/continental/automotive/general/chassis/safety/hidden/elektr_stabilitaetskontrolle_en.html`. Viitattu 26.11.2012.

[3] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems: Concepts and Design*, fourth ed. Addison-Wesley, 2005.

[4] Future Technology Devices International Ltd. An232b-04 data throughput, latency and handshaking, 2006. `http://www.ftdichip.com/Support/Documents/AppNotes/AN232B-04_DataLatencyFlow.pdf`. Viitattu 22.4.2013.

[5] Future Technology Devices International Ltd. Software application development d2xx programmer's guide, 2012. `http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer%27s_Guide(FT_000071).pdf`. Viitattu 12.11.2012.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] Hanspeter Mössenböck, University of Linz. The compiler generator coco/r user manual, 2010. `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Doc/UserManual.pdf`. Viitattu 8.12.2012.

[8] Koskimies, K., and Mikkonen, T. Ohjelmistoarkkitehtuurit, 2005. `http://www.cs.tut.fi/~kk/Ohjelmistoarkkitehtuuri.pdf`. Viitattu 6.2.2013.

[9] Liang, S. Java native interface programmer?s guide and specification, 1999. `http://192.9.162.55/docs/books/jni/download/jni.pdf`. Viitattu 13.1.2013.

[10] Murata Electronics Oy. Configuration and control sw interface specification, 2006. `http://www.muratamems.fi/sites/default/files/uploads/sca3000_demo_kit_macro_language_description_8259200.04.pdf`. Viitattu 12.11.2012.

[11] Murata Electronics Oy. Scc1300 gyro-combo demo kit user manual, 2009. `http://www.muratamems.fi/sites/default/files/documents/scc1300_gyro-combo_demo_kit_user_manual_82117600.01.pdf`. Viitattu 28.11.2012.

[12] Murata Electronics Oy. Vti:n gyroskoopilla voi mitata maapallon pyörimisen, 2011. `http://www.muratamems.fi/fi/uutiset/tiedotteet/vtin-gyroskoopilla-voi-mitata-maapallon-pyorimisen`. Viitattu 26.11.2012.

[13] Murata Electronics Oy. Scc1300-d04 datasheet, 2012. `http://www.muratamems.fi/sites/default/files/documents/82113100d_scc1300-d04_datasheet1.pdf`. Viitattu 23.11.2012.

[14] Murata Electronics Oy. Sensor applications, 2012. `http://www.muratamems.fi/en/applications`. Viitattu 13.11.2012.

[15] Oracle. Java se monitoring and management guide. `http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html`. Viitattu 25.4.2013.

[16] Oracle. Java web start guide. `http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/developersguide/contents.html`. Viitattu 24.4.2013.

[17] Oracle. Java native interface specification, 2012. `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`. Viitattu 12.11.2012.

[18] Rantanen, S. Kiihtyvyysanturien soveltuvuus henkilövaakaan integroituihin ballistokardiografisiin mittauksiin. Master's thesis, Elektroniikan laitos, Aalto yliopisto Sähkötekniikan korkeakoulu, Espoo, Suomi, 2012. `http://lib.tkk.fi/Dipl/2012/urn100721.pdf`.

[19] VESTERHOLM, M., AND KYPPÖ, J. *Java-ohjelmointi*, 6. painos ed. Talentum Media Oy, 2006.

[20] W3C. Document object model (dom) level 3 core specification, 2004. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`. Viitattu 16.4.2013.

[21] W3C. Extensible markup language (xml) 1.0 (fifth edition), 2008. `http://www.w3.org/TR/xml/`. Viitattu 12.12.2012.